2

CR-6-720

ADVANCED SOFTWARE QUALITY ASSURANCE

FINAL REPORT

by

S. H. Saib
D. M. Andrews
J. P. Benson
N. B. Brooks
R. A. Melton

MARCH 1977

GENERAL RESEARCH CORPORATION
P.O. BOX 3587, SANTA BARBARA, CALIFORNIA 93105

Prepared for
THE BALLISTIC MISSILE DEFENSE ADVANCED TECHNOLOGY CENTER

SPONSORED BY

THE BALLISTIC MISSILE DEFENSE ADVANCED TECHNOLOGY CENTER
Under Contract DASG60-76-C-0050
CRDL Item A004

The findings of this report are not be construed as an official Department of the Army position

In addition to approval by the Project Leader and Department Head, General Research Corporation reports are subject to independent review by a staff member not connected with the project. This report was reviewed by Dennis Cooper.

Distribution limited to U.S. Government Agencies only, Test and Evaluation, 17 September 1975. Other requests for this document must be referred to Ballistic Missile Defense Advanced Technology Center, ATTN: ATC-P, P.O. Box 1500, Huntsville, AL 35807.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER CR-6-720 ✓	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
6 Advanced Software Quality Assurance,
~~Final Report~~

5. TYPE OF REPORT & PERIOD COVERED
9 Final Report,

6. PERFORMING ORG. REPORT NUMBER
14 CR-6-720

7. AUTHOR(s)
10 S. H./Saib, N. B./Brooks
D. M./Andrews, R. A./Melton
J. P./Benson,

8. CONTRACT OR GRANT NUMBER(s)
15 DASG60-76-C-0050

9. PERFORMING ORGANIZATION NAME AND ADDRESS
General Research Corporation ✓
P.O. Box 3587
Santa Barbara, CA 93105

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
12 265p.
6.33.04.A.

11. CONTROLLING OFFICE NAME AND ADDRESS
Ballistic Missile Defense Advanced Technology Ctr
P.O. Box 1500
Huntsville, AL 35807

12. REPORT DATE
11 Mar[redacted]77

13. NUMBER OF PAGES
266

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution limited to U.S. Government Agencies only, Test and Evaluation 17 September 1975. Other requests for this document must be referred to Ballistic Missile Defense Advanced Technology Center, ATTN: ATC-P, P.O. Box 1500 Huntsville, AL 35807

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Quality — Symbolic Execution
Software Reliability — IFTRANS
Assertions — Verifiable PASCAL
Verification Conditions — Software Quality Laboratory
Program Proving — Concurrent PASCAL

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is a report of the work performed by General Research Corporation during the Advanced Software Quality Assurance contract. Research was conducted in four major areas: 1) Assertions, verification conditions and consistency proof, 2) Design of a executable assertion language and the development of a preprocessor to implement this language, 3) Development of a Software Quality Laboratory and, 4) Evaluation of existing languages as applied to BMD software problems.

(Continued)

402 754

(Block 20 continued)

This report gives a methodology for verifying software. The preprocessors which allow assertions to be placed in FORTRAN and PASCAL programs are described. The static analysis developed as part of the Software Quality Laboratory are also described and examples are given of their use. The Concurrent PASCAL programming language is applied to a generic model of a BMD software system.

ABSTRACT

This is a report of the work performed by the General Research Corporation during the Advanced Software Quality Assurance contract. Research was conducted in four major areas:

1. Assertions, verification conditions and consistency proof
2. Design of an executable assertion language and the development of a preprocessor to implement this language
3. Development of a Software Quality Laboratory
4. Evaluation of existing languages as applied to BMD software problems

This report gives a methodology for verifying software. The preprocessors which allow assertions to be placed in FORTRAN and PASCAL programs are described. The static analyses developed as part of the Software Quality Laboratory are also described and examples are given of their use. The Concurrent PASCAL programming language is applied to a generic model of a BMD software system.

CONTENTS

CONTENTS (Contd.)

ILLUSTRATIONS

ILLUSTRATIONS (Contd.)

ILLUSTRATIONS (Contd)

1 INTRODUCTION

In the current contract, GRC's work in advanced quality assurance was concentrated on developing a capability for the formal verification of large real-time systems. It was found in the course of this work that some techniques useful in the formal verification process are also applicable to the problem of detecting errors arising from erroneous sensor data or hardware failures. In addition, languages that appear to promise higher software quality were evaluated.

1.1 BACKGROUND

This effort was an outgrowth of an initial study, the Reliable Software Study,[1,2] which suggested a number of techniques that could lead to improved software. Emphasis has been placed on techniques that originated with Floyd,[3,4] were shown to be useful by King,[5] and further developed by others.[6-9] Specific costly errors, such as those described by Osterweil and Fosdick[10] and in a study by Logicon,[11] were also addressed.

Since the effort was directed towards the verification of real-time programs, it was decided that the target languages should be "real." By real, it is meant that a compiler exists for the language such that a program written in it can execute normally on a machine. This is a strong criterion that most others have not chosen to meet. The target languages were FORTRAN and PASCAL. As far as the techniques are concerned, their analysis appears very similar, requiring only separate front ends. Each of the languages has a dialect--IFTRAN for FORTRAN and Verifiable PASCAL for PASCAL. Preprocessors were developed to generate standard FORTRAN from IFTRAN and standard PASCAL[12] from Verifiable PASCAL. The resulting programs execute on computers such as the CDC 6400 and CDC 7600 which have standard compilers.

The system that actually performs the analysis was built using IFTRAN.

1.2 ACCOMPLISHMENTS

GRC's work has resulted in the establishment of the Software Quality Laboratory. This facility contains tools which can be used to detect common semantic errors, assist in testing, formally verify computer programs, and provide for fault tolerance.

The facility has been designed to analyze both FORTRAN and PASCAL. Extensions to both these languages have been designed and implemented so that common errors can be detected and formal verification is possible.

The languages with their extensions have been translated into executable code. The redundancy present in the executable extensions provides for the detection of errors during program execution. This capability will be used in future work in fault tolerance.

Since one of the goals of this work has been to apply formal verification to practical programs, verification condition generators were implemented for both languages. The generators handle logical, fixed point, character, and floating-point data types. Multidimension arrays are presently handled and PASCAL records will be handled in the near future.

The verification process divides programs into very small segments each of which is verified independently of all other parts. Each verification condition is tightly coupled to the code segment with which it is associated. Both the code segment and the verification condition can be listed on a printer and/or displayed on an interactive terminal.

A simplifier which contains many standard simplification rules can be invoked to cause automatic simplification of verification conditions.

Rules which are not in the simplifier can be applied to individual verification conditions. These rules can be text replacement rules or pattern matching rules. They can be applied only once or saved as an axiom to be used again.

We have also looked at the classes of concurrency present in a BMD system and at the possibility of using Co·current PASCAL as a language for a BMD software system. The advantages and disadvantages of Concurrent PASCAL for this application have been noted.

1.3 REPORT ORGANIZATION

Section 2 outlines how to prepare software for verification. It contains an overall description of the Software Quality Laboratory, with some examples of how the various parts can be used to validate software. It also contains a set of recommendations for the development of quality software. These have resulted from our experience with the techniques discussed in this report.

Section 3 describes the language extensions that were made to FORTRAN and PASCAL so that the software written in those languages could be verified. Section 3 emphasizes the translation of the extensions into executable code. These same languages are the ones for which the static analysis tools and formal verification techniques a:e available.

Section 4 describes techniques that can be used to remove errors before a complete verification can be completed. These techniques could be implemented in a super-compiler. However, they were implemented in a separate process which allowed this work not to duplicate the compiler, allowed the assumption that syntax errors had been removed, and provided data which was used later in the formal verification process.

Section 5 shows how quality can be improved in the testing process through the use of coverage tests and executable assertions. Also included is a discussion as to how the executable assertions can be used to derive loop invariants and to detect faults at execution time.

Section 6 is concerned with the formal verification process. Once software has been shown to be free of syntax and major categories of semantic errors using the techniques described in previous sections, it is ready for formal verification. This section describes the process of verification from the design and implementation of the verification condition generator and simplifier to examples of validated programs.

Section 7 discusses the possible application of Concurrent PASCAL to a large software system. The advantages of and disadvantages of Concurrent PASCAL are demonstrated with an example of its application to a real-time simulation program. The types of concurrency which are present in a BMD system are presented, and assertions for such programs are given.

Appendixes are provided to describe the formal grammar of Verifiable PASCAL (Appendix A), the translation templates for the generation of PASCAL from Verifiable PASCAL (Appendix B), the translation templates for the generation of FORTRAN from IFTRAN with assertions (Appendix C), and the formal verification of sample programs (Appendix D).

2 METHODOLOGY FOR VERIFICATION

As the Software Quality Laboratory evolved, it became obvious that it was not possible to verify software that had not been written with verification in mind. This section outlines how software can be prepared for the verification process, shows what can be accomplished through several stages of verification, and how the tools available for proofs of correctness can be used to provide designed-in quality. The various steps in this verification procedure are shown in Fig. 2.1.

2.1 RECOMMENDATIONS

Based upon GRC's experience to date in the development and use of the Software Quality Laboratory, the following recommendations, which should lead to higher quality software, are made:

1. Use small modules. Designs that use small modules with as few global variables and paths as possible will aid the verification process.

2. State data access rights. Every global variable should be declared either an input, output, or both.

3. Limit variable rights. When computationally feasible, keep inputs separate from outputs.

4. State units. Every variable should have its units declared.

5. State ranges. Every variable should have its range of values declared at module entry and exit.

6. State invariants. When a relation between variables is known to be always true, declare that relation. Try to design algorithms that lend themselves to relations that are always true rather than almost always true.

7. Place constraints on results. Every output variable should be bound as closely as possible to an expression in terms of the input variables.

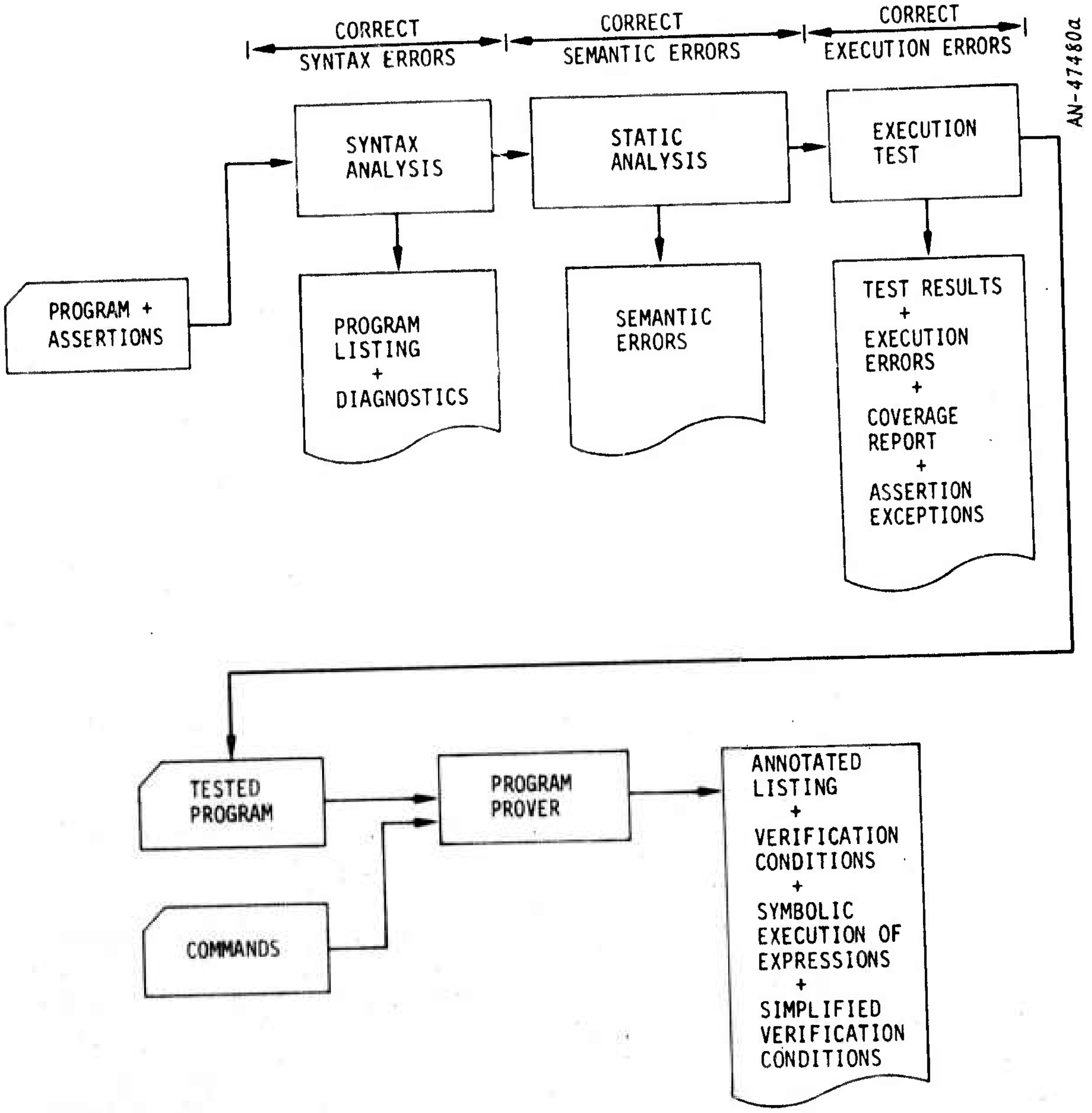

CORRECT
SYNTAX ERRORS
CORRECT
SEMANTIC ERRORS
CORRECT
EXECUTION ERRORS
AN-47480a
SYNTAX ANALYSIS
STATIC ANALYSIS
EXECUTION TEST
PROGRAM + ASSERTIONS
PROGRAM LISTING + DIAGNOSTICS
SEMANTIC ERRORS
TEST RESULTS + EXECUTION ERRORS + COVERAGE REPORT + ASSERTION EXCEPTIONS
TESTED PROGRAM
PROGRAM PROVER
COMMANDS
ANNOTATED LISTING + VERIFICATION CONDITIONS + SYMBOLIC EXECUTION OF EXPRESSIONS + SIMPLIFIED VERIFICATION CONDITIONS

Figure 2.1. Steps in Verification Procedure

8. Use tools to eliminate errors.

   a. Use static analysis tools to eliminate unreasonable errors before testing single modules.

   b. Make sure the execution test step checks each combination of paths in a module before integration.

   c. Use static analysis tools to check module interfaces before testing modules together.

   d. Turn on executable assertions during execution test.

9. Perform formal verification. Use assertions developed during testing for formal verification.

2.2 VERIFICATION WITHOUT REDUNDANCY

The key characteristic of verifiable software is redundant information that is provided to allow a checking process between the redundant information and the software itself. This is not unlike the systems used in accounting which use double entries to provide a check. If there is no redundant information, there is little that can be done to verify the software. What is normally done instead is to run a set of test cases for which answers are known. Unfortunately, the testing approach cannot hope to completely verify large programs for there are far too many possible combinations of paths through them and too many possible combinations of variables to ever test all paths with all values. Since existing systems have not been built with verifiable software, we have the situation in large existing programs where 54% of the errors are found after acceptance test and 50% of the life cycle cost has been assigned to the time period following acceptance tests.[13] This time period, which is often named the "maintenance" phase of the project, does not "maintain" the software in its delivered state, but in fact tries to fix the errors that are found after the software should be error-free.

There are a few valuable checks that can be made without redundant information. The only check currently in use by most developers of large systems is the syntax check which checks that every statement in the program satisfies the rules of the programming language used. Statements that do not satisfy the rules are designated as being in error, and presented, along with the program listing, in a report known as the diagnostic report. It is assumed that the programmer will correct these problems before proceeding. As a result, most large systems today prevent the programmer from testing a program before removing the syntax errors. Since syntax errors are so easy to detect at an early stage in the program development, they are not regarded as serious or costly. The Software Quality Laboratory has taken the approach that other errors (due to an unreasonable sequence of statements) should also be reported to the programmer. This would be done in a fashion similar to the diagnostics report so that these simple (but often costly) errors can be removed at the same time as the syntax errors and with as little effort. The main difference between the syntax errors detected by a language processor and the simple, unreasonable errors found by the Software Quality Laboratory is that a sequence of statements must be examined rather than a single statement. These errors are then reported in terms of the groups of statements that could have caused the error, rather than a single statement.

The unreasonable errors that do not require redundant information are the set/use errors, mode errors, infinite loop errors, external reference errors, and unreachable code errors. The method of detecting and reporting each of these errors is fully described in Sec. 4. This section briefly introduces the analysis that is performed by means of examples. The unreasonable errors are removed in the step shown in Fig. 2.1 labeled STATIC ANALYSIS. This step is done after the syntax analysis.

2.2.1 Set/Use Errors

Two types of set/use errors are detected. One of the most common is use of a variable before it is set. This is usually due to forgetting to initialize the variable. The other is that the variable is set, but is not used. This last error is often due to a misspelling of the variable. This brief program demonstrates some common set/use errors in IFTRAN and PASCAL.

```
SUBROUTINE AVER(A, N, ANS)
      INTEGER N, I, J
      REAL A(1), ANS, SUM
      J = 1
      WHILE (I .LE. N)
            SUM = SUM + A(I)
            I = I + 1
      END WHILE
      ANS = SUM/FLOAT(N)
      RETURN
END
```

IFTRAN Set/Use Errors

```
PROCEDURE aver (VAR a : ARRAY[1..n] OF real;
      VAR ans : real);
VAR i, j : integer;
      sum : real;
BEGIN
      j := 1;
      WHILE i <= n DO
            sum := sum + a[i];
            i := i + 1
      END WHILE;
      ans := sum/n
END;
```

V-PASCAL Set/Use Errors

The program shows three set/use errors that will be detected by the Software Quality Laboratory. The variables I and SUM are not initialized, although they are used in a WHILE loop, and the variable J is set but not used.

2.2.2 Mode Errors

If we had the statement

ANS = SUM/N

rather than

ANS = SUM/FLOAT(N)

a mode warning would be produced as well, since real and integer variables should not appear in the same expression. This error is more important in the multi-module case where one routine interfaces with another, as in these programs.

```
   PROGRAM STAT(INPUT, OUTPUT)
         INTEGER DATA(10), I, M, N
         READER = 5
         PRIN = 6
         N = 10
         READ (READER, 1)(DATA(I), I = 1, 10)
1        FORMAT(I5)
         CALL AVER(DATA, N, RESULT)
         WRITE(PRIN, 2) RESULT
2        FORMAT(F6.2)
         STOP
   END
```

IFTRAN Multi-Module Mode Error

```
PROGRAM stat(input, output);
CONS n = 10;
VAR data : ARRAY[1..n] OF integer;
      result : real;
      i, m : integer;
      (*PROCEDURE aver goes here*)
BEGIN
      FOR i := 1 TO n DO
            read(input, data[i])
      END FOR;
      aver(data, result);
      write(output, result)
END.
```

V-PASCAL Multi-Module Mode Error

The program STAT invokes AVER to calculate the average of ten data values. Unfortunately, AVER expects that the data is real while STAT provides integer data. The result, of course, would be erroneous if the program were ever executed.

One can argue that some modern compilers, notably PASCAL, provide this facility for mode checking. To do so they require that every module be compiled together. As a result, a program with 10^5 lines of code has not been written using such languages. The static analysis facility is concerned with the analysis of systems with very large programs (e.g., 10^6 lines of code), hence it was believed important to provide checking of external modules after syntax errors had been removed.

2.2.3 External Reference Errors

Another costly error which is detectable without redundant information is the misuse of external routines with the incorrect number of parameters. If, for example, the IFTRAN CALL statement had been

```
CALL AVER (DATA, N, RESULT, FLAG)
```

or the V-PASCAL statement had been

```
aver (data, result, flag);
```

the error report would have indicated that the external routine did not have the same number of parameters.

2.2.4 Infinite Loop Errors

Infinite loops often result because on one path the control variable to exit the loop was not modified. If this is the case, an error report listing the problem loop is given, as shown below.

```
WHILE (M .LT. N)                    WHILE m < n DO
     I = (M + N)/2                       i := (m + n) DIV 2;
     IF (X .LT. A(I))                    IF x < a[i] THEN
          N = I                               n := I
     ORIF (X .GT. A(I))                  ORIF x > a[i] THEN
          M = I                               m := I
     ELSE                                ELSE
          LOOKUP = I                          lookup := i
     ENDIF                               ENDIF
END WHILE                           END WHILE
```

IFTRAN Infinite Loop Error   V-PASCAL Infinite Loop Error

Once entered, the above loop would not exit if X is equal to some element in the array A between A(M) and A(N) since neither M nor N is modified once that element is found.

2.2.5 Unreachable Code Errors

Another costly error is unreachable code. This is costly because the expense of designing, preparing, and storing the code could have been avoided if it is not needed. On the other hand, if it is meant to execute under some circumstances, then the fact that it cannot be reached is an error. Structurally unreachable code is most common in large unstructured programs with unconditional transfers. In the example shown below, statement 300 is unreachable.

```
          REAL FUNCTION DISCR(A, B, C)
                REAL A, B, C, D
                D = A**2 - 4.0*B*C
                IF (D .LT. 0.0)
                      GO TO 100
                ELSE
                      GO TO 200
                END IF
300             DISCR = 0.0
                GO TO 400
100             DISCR = -D
                GO TO 400
200             DISCR = D
400             RETURN
                END
```

IFTRAN Unreachable Code

```
          FUNCTION discr (a, b, c : real) : real;
                LABEL 100, 200, 300, 400;
                VAR d : real;
                BEGIN
                      d := a*a - 4.0*b*c;
                      IF d < 0.0 THEN
                            GOTO 100
                      ELSE
                            GOTO 200
                      END IF;
300:                  discr := 0.0;
                      GOTO 400;
100:                  discr := -d;
                      GOTO 400;
200:                  discr := d
400:            END;
```

V-PASCAL Unreachable Code

2.3 VERIFICATION WITH REDUNDANCY

While the preceding errors can be found without the need for redundant information, the errors most difficult to find require additional information specifically for program verification. The elements of this information are known collectively as assertions. Every assertion added to a program allows checks to be performed that can eliminate difficult-to-find errors. There are several forms of assertions. Some are relatively easy to state and are merely the concrete expression of a designer's normal thoughts about a program. Some are more difficult to state because designers do not normally take into account all the possible values that a variable can take on. Finally, some are very difficult to state because they require a statement of the assumptions upon which a design is based.

Primarily because some assertions are more difficult to state than others, the Software Quality Laboratory is able to accept a partially asserted program and use these to detect some errors. This is similar to the early detection of hardware errors where, due to the difficulty of a complete check, parity checks are made on memories assuming only a single bit per word could be in error.

2.3.1 Asserted/Actual Use

One of the easiest assertions to state is one of the most powerful in checking the correct usage of variables between modules. A common error is the use of a variable as an input to a routine when it is an output, or vice versa. In languages where a large number of variables are said to be "known" to the program, as in FORTRAN with its COMMON blocks or in PASCAL with its global variables, errors in the misuse of variables are costly to find.

In order to aid in checking that each variable is used correctly, the INPUT,OUTPUT assertions were designed and implemented both to provide an error detection capability before execution of the program and a trace of the input and output variables during test runs. The error detection capability is further discussed in Sec. 4 and the trace capability is discussed in Sec. 3. Basically, what is checked is how the assertion that states how a variable is to be used in a given module matches with its actual use. For example, consider a routine TIMES that multiplies A and B together and stores them in C:

```
          SUBROUTINE TIMES (A, B, C)
                INTEGER A, B, C
COMMENT *** INPUT ASSERTION ***
                INPUT (/INTEGER/A, B)
                C = 0
                WHILE (B .GT. 0)
                      C = C + A
                      B = B - 1
                END WHILE
COMMENT *** OUTPUT ASSERTION ***
                OUTPUT (/INTEGER/C)
                RETURN
          END
```

IFTRAN Input/Output Error

```
PROCEDURE times (VAR a, b, c : integer);
      BEGIN
      (*** INPUT ASSERTION ***)
            INPUT a, b;
            c := 0;
            WHILE b > 0 DO
                  c := c + a;
                  b := b - 1
            END WHILE;
      (*** OUTPUT ASSERTION ***)
            OUTPUT c
      END;
```

V-PASCAL Input/Output Error

An error would be reported since the value of B is changed. Hence B is not just an input, it is also an output.

More important is the capability for checking that variables are used correctly across modules. In the following example, the variable ANS is asserted to be used both as an INPUT and as an OUTPUT. It is intended that the CALL will compute ANS times F and store the result in ANS, but because the TIMES module expects ANS to be used only as an INPUT as the first parameter, and only as an OUTPUT as the third parameter, errors would be reported. In this example, note that since TIMES sets the OUTPUT variable C to zero, ANS would also be set to zero and the result would be zero. Such errors can be extremely difficult to find if several levels of a subroutine hierarchy are involved.

```
          INTEGER ANS, F
                .
                .
                .
    COMMENT *** INPUT ASSERTION ***
          INPUT (/INTEGER/ANS, F)
                .
                .
                .
          CALL TIMES (ANS, F, ANS)
                .
                .
                .
    COMMENT *** OUTPUT ASSERTION ***
          OUTPUT (/INTEGER/ANS)
                .
                .
                .
```

IFTRAN Input/Output
Multimodule Error

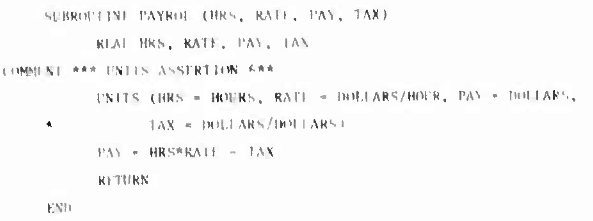
```
          ans, f : integer;
                .
                .
                .
    (*** INPUT ASSERTION ***)
          INPUT ans, f;
                .
                .
                .
          times (ans, f, ans);
                .
                .
                .
    (*** OUTPUT ASSERTION ***)
          OUTPUT ans;
                .
                .
                .
```

V-PASCAL Input/Output
Multimodule Error

2.3.2 Units Consistency

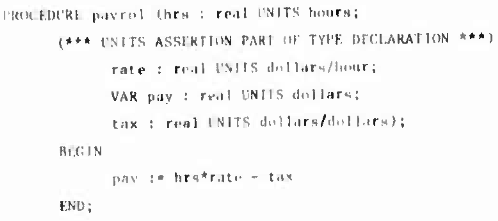
Another simple-to-state but very powerful assertion is the UNITS assertion which is used to check that units are consistent throughout a program. This is further discussed in Sec. 4, but a brief example is appropriate here.

The units check is one that everyone trained in the physical sciences has been urged to apply. Basically, it prevents apples being assigned to oranges, miles being assigned to feet, dollars being added to pounds, and other similar but troublesome errors, as demonstrated here:



```
    SUBROUTINE PAYROL (HRS, RATE, PAY, TAX)
          REAL HRS, RATE, PAY, TAX
COMMENT *** UNITS ASSERTION ***
          UNITS (HRS = HOURS, RATE = DOLLARS/HOUR, PAY = DOLLARS,
    *           TAX = DOLLARS/DOLLARS)
          PAY = HRS*RATE - TAX
          RETURN
    END
```

IFTRAN Units Error



```
PROCEDURE payrol (hrs : real UNITS hours;
      (*** UNITS ASSERTION PART OF TYPE DECLARATION ***)
            rate : real UNITS dollars/hour;
            VAR pay : real UNITS dollars;
            tax : real UNITS dollars/dollars);
      BEGIN
            pay := hrs*rate - tax
      END;
```

V-PASCAL Units Error

Because the UNITS of TAX do not match the UNITS of the other terms in the expression, an error would be reported. One can regard the UNITS assertion as a very strong type declaration. Designers are accustomed to specifying the type of arithmetic that their machine is using. Many recognize the dangers of mixed mode arithmetic, and the advantages of checking to verify that only one kind of arithmetic is actually used in an expression. However, here the dangers of allowing a variable named MONEY expressed in dollars at one time and at another time being expressed in cents have been overlooked. In IFTRAN a separate UNITS statement is used for the UNITS assertion. In V-PASCAL, the UNITS assertion is part of the type declaration.

All the errors and assertions discussed to this point can be found once syntax analysis by a compiler has been completed, and before the program is submitted for an execution test.

2.3.3 Logical Assertions

There are additional assertions which can be used in the execution test (see the EXECUTION step in Fig. 2.1) and which can also be used in a correctness proof.

The easiest to use of the additional assertions is the INITIAL assertion. The INITIAL assertion, which is more fully described in Secs. 3 and 5, is meant to provide a means for explicitly stating the initial conditions on variables. This is most often a limitation on the range that a variable can take on. For example, the TIMES program presented in this section would not operate correctly unless $B \geq 0$. This condition can be stated in the INITIAL assertion by placing, as the first executable statement in the program,

```
INITIAL (B .GE. 0)                  INITIAL b >= 0;

      IFTRAN                           V-PASCAL
```

which will ensure that on every entry to the routine, the second argument is not less than zero.

One might also note that the TIMES program would not work correctly if the result of A*B were too large for the machine. This error, known as fixed-point overflow, is not detected on many machines and as a result can be a costly error to locate. One way to state an initial assertion to prevent undetected overflow in the TIMES program would be to ensure that the sum of the powers of each of the input values is less than the largest number that can be represented in the machine. We would then represent the assertion by

```
IFTRAN
      INITIAL (B .GE. 0 .AND. POWER(A) + POWER(B) .LT. MAXP)

V-PASCAL
      INITIAL b >= 0 AND power(a) + power(b) < maxp;
```

For example, in a small computer with only 16-bit words, MAXP would be 15. In a computer with 60-bit words, MAXP would be 59.

POWER is a function that returns which power of 2 is greater than or equal to the input value

$$2^{POWER} \geq A > 2^{POWER-1}$$

Any time that an assertion is not true, an exception report is printed stating which assertion in which module is in error. Provision has also been made for allowing a block of code to be invoked in case an assertion is not true. The block of code can be used for example to correct the condition that caused the exception or to print out more pertinent error messages.

By placing additional conditions on input variables, more checks can be made that prevent errors. When there is no condition stated on an input variable, it is equivalent to stating that any value from the smallest possible represented by the machine to the largest possible will not cause an error. Since this is a very unlikely situation, thought should be given to just what conditions are being assumed by the designer. For example, in a particular payroll program it might be known that a person can never work more than 168 hours between checks, that the pay rate can never be more than $25.00 per hour, and that the tax rate can never be more than 50%. We could state this in an assertion as

```
IFTRAN
          INITIAL (HOURS .LE. 168 .AND. RATE .LE. 25.00 .AND. TAX
                   .LT. 0.50)

V-PASCAL
          INITIAL hours <= 168 AND rate <= 25.00 AND tax < 0.50;
```

if the input variables were HOURS, RATE, and TAX. This would help prevent errors such as might result from reading or punching a time card incorrectly.

Besides stating assertions on the input variables with an INITIAL assertion, it is also possible to state assertions on output variables with a FINAL assertion. As with the INITIAL assertion, the FINAL assertion can also be used to check on ranges of output variables such as in a payroll program to check that an employee is not paid too much or too little,

```
IFTRAN
          FINAL (PAY .LT. 500.00 .AND. PAY .GT. 40.00)
V-PASCAL
          FINAL pay < 500.00 AND pay > 40.00;
```

or in a flight control program to check that an airplane is not too high or too low

```
IFTRAN
          FINAL (HEIGHT .LT. MAXALT .AND. HEIGHT .GT. MINALT)
V-PASCAL
          FINAL height < maxalt AND height > minalt;
```

or in a chemical processing program to check that the calculated temperature denotes the liquid state

```
IFTRAN
          FINAL (TEMP .LT. BOIL .AND. TEMP .GT. FREEZ)
V-PASCAL
          FINAL temp < boil AND temp > freez;
```

or in an airline reservation system to denote that the number of seats remaining is reasonable

```
IFTRAN
          FINAL (SEATS .LT. 425 .AND. SEATS .GE. 0)
V-PASCAL
          FINAL seats < 425 AND seats > 0;
```

It is better to state acceptable ranges of output variables than not to state anything about a variable so that obvious problems can be detected. However, it is best if possible to state the results of the module in terms of the input variables so that the assertion can be used in a formal verification. This is not very difficult to do if, for example, a series approximation is being used and the error term is known or if the inverse function is known.

For example, if the module computed the square root of an input variable x and the result was in y, the final assertion could be:

```
IFTRAN
        FINAL (Y**2 .EQ. X)

V-PASCAL
        FINAL y*y = x;
```

which uses the operation of raising to a power of 2 as the inverse operation of square root.

In the TIMES module given previously the final assertion could be:

```
IFTRAN
        FINAL (C .EQ. A*B)

V-PASCAL
        FINAL c = a*b;
```

A check on the calculation of the altitude of a plane might be

```
IFTRAN
        FINAL (ALT**2 + GND**2 .EQ. RANGE**2)

V-PASCAL
        FINAL alt*alt + gnd*gnd = range*range;
```

The object of a FINAL assertion is to place as tight a condition on an output variable as possible. In the examples shown, an equality relation was used. This is the strongest relation that can be used. However, in other cases a bound is expressed. For example, it might be necessary in a floating point machine to state a relative bound on a result such as

```
IFTRAN
        FINAL (Y**2 - X .LT. 3.0E - 10*X)
```

```
V-PASCAL
          FINAL y*y - x < 3.0e - 10*x
```

where 3.0e - 10*x is the error bound of the algorithm.

The FINAL assertion is meant to provide a facility for specifying a program's function. When it thoroughly specifies all the outputs of the routine, it can be used in a formal program verification as described in Sec. 6. When it partially specifies the outputs, it can be used during execution test as described in Sec. 5 for fault detection and assertion refinement.

3 LANGUAGE EXTENSIONS AND IMPROVEMENTS

The Statement of Work requirement to produce verifiable software written in FORTRAN or PASCAL has been met by GRC, in part, by improving the readability of the code and by adding features to support automated verification. This section describes a number of changes which respond to these needs. The language changes fall into several categories:

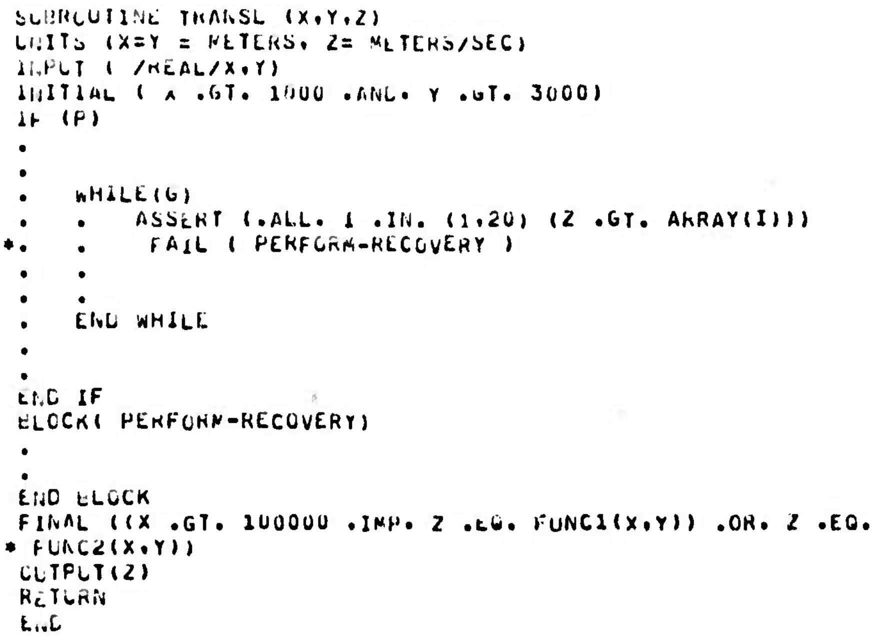
- Improved control structures, which are easier to write and which result in more readable code
- Executable assertion statements (ASSERT, INITIAL, FINAL), which may be used to report assertion exceptions during testing and can also be used by a program verifier
- Data access statements (INPUT, OUTPUT), which qualify or limit the access rights and operations on data by explicitly specifying the input and output variables
- Unit qualifiers for variables, which declare the physical units in addition to any type declarations, thereby making units consistency checking possible

These capabilities are implemented with preprocessors which accept as input source code written in an extended language (IFTRAN or V-PASCAL) and generate a standard language (FORTRAN or PASCAL) for compilation. Figure 3.1 is sample IFTRAN program showing the assertions and some control constructs. A portion of a PASCAL program that uses some of the language enhancements is shown in Fig. 3.2.

3.1 IMPROVED CONTROL STRUCTURES

3.1.1 IFTRAN Control Constructs

Unlike PASCAL and ALGOL-based languages (in which complex statements are built in terms of decision statements and BEGIN...END clauses), IFTRAN control constructs are composed of readily identifiable and



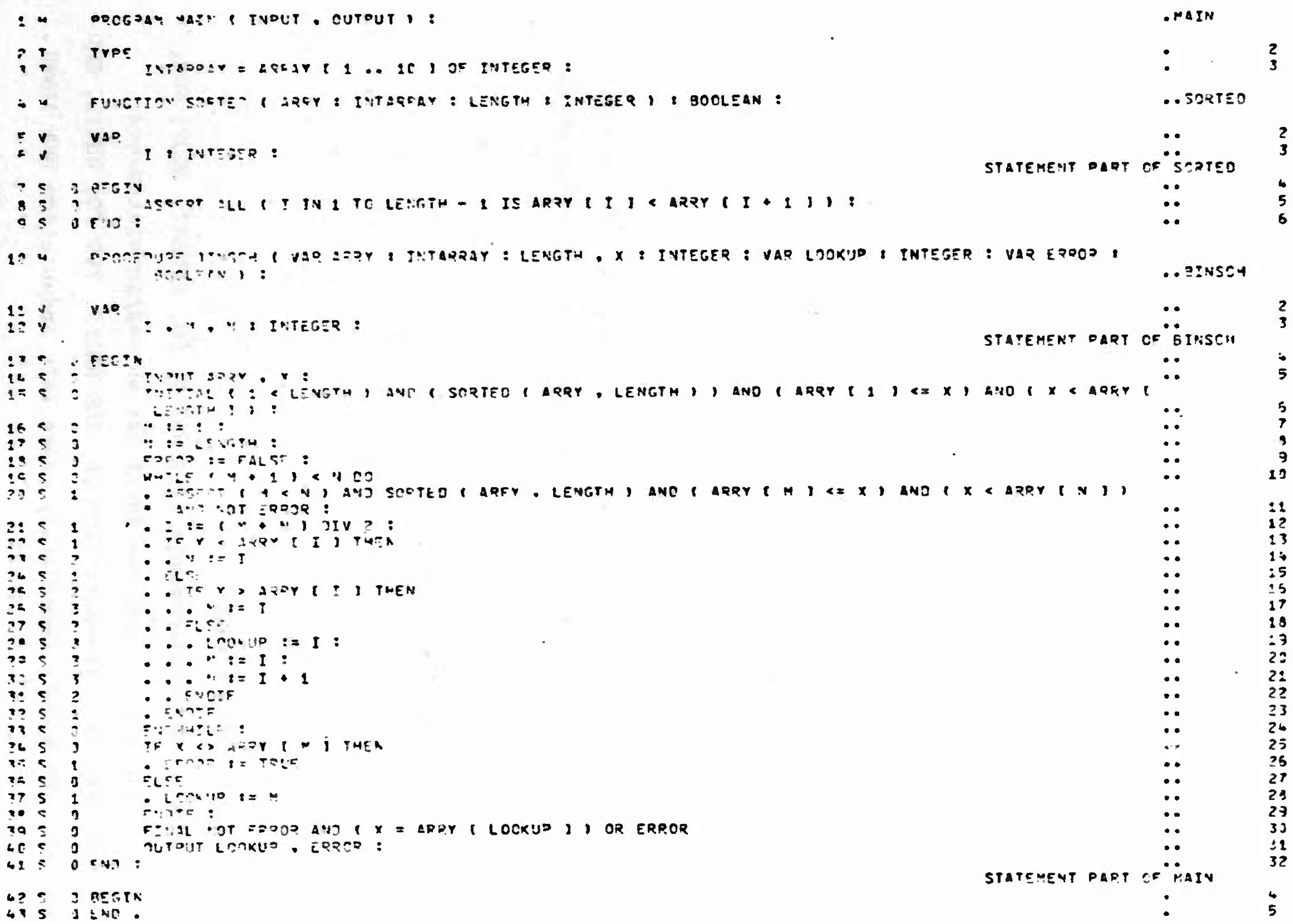
```
SUBROUTINE TRANSL (X,Y,Z)
UNITS (X=Y = METERS, Z= METERS/SEC)
INPUT ( /REAL/X,Y)
INITIAL ( X .GT. 1000 .AND. Y .GT. 3000)
IF (P)
.
.
.   WHILE(Q)
.   .    ASSERT (.ALL. I .IN. (1,20) (Z .GT. ARRAY(I)))
*.  .     FAIL ( PERFORM-RECOVERY )
.   .
.   .
.   END WHILE
.
.
END IF
BLOCK( PERFORM-RECOVERY)
.
.
END BLOCK
FINAL ((X .GT. 100000 .IMP. Z .EQ. FUNC1(X,Y)) .OR. Z .EQ.
* FUNC2(X,Y))
OUTPUT(Z)
RETURN
END
```

Figure 3.1. Example Showing Use of IFTRAN Language Extensions

matching control statements. IFTRAN constructs are defined by pairs of beginning and ending IFTRAN control statements. The following pairs are legal:

IF...END IF
WHILE...END WHILE
DO...END DO
REPEAT...UNTIL
LOOP...END LOOP
FOR...END FOR
BLOCK...END BLOCK

Beginning IFTRAN control statements (IF, WHILE, DO, REPEAT, LOOP, FOR, BLOCK) cause right indentation one level for succeding statements. Ending IFTRAN control statements (END IF, END WHILE, END DO, UNTIL, END LOOP, END FOR, END BLOCK) immediately cause left indentation one level.



```
 1 M     PROGRAM MAIN ( INPUT , OUTPUT ) ;                                                              .MAIN

 2 T     TYPE                                                                                           .          2
 3 T          INTARRAY = ARRAY [ 1 .. 10 ] OF INTEGER ;                                                 .          3

 4 M     FUNCTION SORTED ( ARRY : INTARRAY ; LENGTH : INTEGER ) : BOOLEAN ;                             ..SORTED

 5 V     VAR                                                                                            ..         2
 6 V          I : INTEGER ;                                                                             ..         3
                                                                                   STATEMENT PART OF SORTED
 7 S   0 BEGIN                                                                                          ..         4
 8 S   0      ASSERT ALL ( I IN 1 TO LENGTH - 1 IS ARRY [ I ] < ARRY [ I + 1 ] ) ;                      ..         5
 9 S   0 END ;                                                                                          ..         6

10 M     PROCEDURE BINSCH ( VAR ARRY : INTARRAY ; LENGTH , X : INTEGER ; VAR LOOKUP : INTEGER ; VAR ERROR :
              BOOLEAN ) ;                                                                               ..BINSCH

11 V     VAR                                                                                            ..         2
12 V          I , M , N : INTEGER ;                                                                     ..         3
                                                                                   STATEMENT PART OF BINSCH
13 S   0 BEGIN                                                                                          ..         4
14 S   0      INPUT ARRY , X ;                                                                          ..         5
15 S   0      INITIAL ( 1 < LENGTH ) AND ( SORTED ( ARRY , LENGTH ) ) AND ( ARRY [ 1 ] <= X ) AND ( X < ARRY [
               LENGTH ] ) ;                                                                             ..         6
16 S   0      M := 1 ;                                                                                  ..         7
17 S   0      N := LENGTH ;                                                                             ..         8
18 S   0      ERROR := FALSE ;                                                                          ..         9
19 S   0      WHILE ( M + 1 ) < N DO                                                                    ..        10
20 S   1      . ASSERT ( M < N ) AND SORTED ( ARRY , LENGTH ) AND ( ARRY [ M ] <= X ) AND ( X < ARRY [ N ] )
              *  AND NOT ERROR ;                                                                        ..        11
21 S   1      . I := ( M + N ) DIV 2 ;                                                                  ..        12
22 S   1      . IF X < ARRY [ I ] THEN                                                                  ..        13
23 S   2      . . N := I                                                                                ..        14
24 S   1      . ELSE                                                                                    ..        15
25 S   2      . . IF X > ARRY [ I ] THEN                                                                ..        16
26 S   3      . . . M := I                                                                              ..        17
27 S   2      . . ELSE                                                                                  ..        18
28 S   3      . . . LOOKUP := I ;                                                                       ..        19
29 S   3      . . . M := I ;                                                                            ..        20
30 S   3      . . . N := I + 1                                                                          ..        21
31 S   2      . . ENDIF                                                                                 ..        22
32 S   1      . ENDIF                                                                                   ..        23
33 S   0      ENDWHILE ;                                                                                ..        24
34 S   0      IF X <> ARRY [ M ] THEN                                                                   ..        25
35 S   1      . ERROR := TRUE                                                                           ..        26
36 S   0      ELSE                                                                                      ..        27
37 S   1      . LOOKUP := M                                                                             ..        28
38 S   0      ENDIF ;                                                                                   ..        29
39 S   0      FINAL NOT ERROR AND ( X = ARRY [ LOOKUP ] ) OR ERROR                                      ..        30
40 S   0      OUTPUT LOOKUP , ERROR ;                                                                   ..        31
41 S   0 END ;                                                                                          ..        32
                                                                                   STATEMENT PART OF MAIN
42 S   0 BEGIN                                                                                          .          4
43 S   0 END .                                                                                          .          5
```

Figure 3.2. Example Showing Use of PASCAL Language Extensions

IFTRAN control statements which must match are connected with vertical dots on the indented IFTRAN source listing:

CORRECTLY NESTED CONSTRUCTS

```
IF ( )
. WHILE ( )
. . DO ( )
. . . statement(s)
. . END DO
. END WHILE
END IF
```

This enhances the visibility of program structure, as well as providing a visual debugging aid. Indentation level must be zero when an END statement is encountered.

An IFTRAN control statement is either an IFTRAN keyword or an IFTRAN keyword followed by a character string contained within balancing parentheses. Proper combinations of IFTRAN control statements form IFTRAN control constructs.

The simplest IFTRAN control statement has the form

IFTRAN-KEYWORD

where IFTRAN-KEYWORD may be ELSE, END IF, END WHILE, REPEAT, END DO, END FOR, LOOP, EXIT, END LOOP, END BLOCK, or END. Blanks within IFTRAN keywords are not significant. Each keyword must be a separate statement.

The second form of an IFTRAN control statement is

IFTRAN-KEYWORD(CHARACTER-STRING)

where IFTRAN-KEYWORD may be IF, ORIF, EXIT IF, WHILE, UNTIL, DO, FOR, INVOKE, or BLOCK. Keyword conventions are the same as for the simpler IFTRAN control statement form. For a statement of this form to be an IFTRAN statement, the left parenthesis following the keyword must be balanced by the last non-blank character of the statement (which must be a right parenthesis). Parentheses within Hollerith strings are not counted.

The form of CHARACTER-STRING is not examined by the IFTRAN preprocessor (except for balancing parenthesis and restrictions in the DO and FOR statements). To be translated into compilable FORTRAN, CHARACTER-STRING should be a FORTRAN logical expression for keywords IF, ORIF, WHILE, UNITL, and EXIT IF. If the embedded language is not FORTRAN, this is not necessary. For example, the embedded language can be English for purposes of program design documentation.

```
IFTRAN STATEMENTS
IF(COMPUTATION IS COMPLETE)
WHILE(FLIGHT 76 IS IN DULLES)
```

IF...ORIF...ELSE...END IF

The IF...ORIF...ELSE...END IF construct provides the ability to select at most one (but possibly none) among several alternate groups of statements to execute. The basic form of this construct is the matching IF...END IF pair. If the FORTRAN EXPRESSION is true, control proceeds to the first statement within the construct; otherwise control transfers to the END IF statement.

Use of the ORIF and ELSE are optional. There may be more than one ORIF condition stated; they will be tested consecutively and, if one of the conditions is true, control will be transfered to the first statement after that ORIF. Otherwise, control proceeds to an ELSE, if one is present, or the END IF. Figure 3.3a shows the Statement Syntax and 3.3b is a Construct Flowchart.

```
IF ( EXPRESSION1 )
.
.   STATEMENTS TO EXECUTE IF EXPRESSION1 IS TRUE
.
ORIF ( EXPRESSION2 )
.
.   STATEMENTS TO EXECUTE IF EXPRESSION 1 IS FALSE AND EXPRESSION 2
.   IS TRUE
.
ORIF ( EXPRESSION N )
.   STATEMENTS TO EXECUTE IF EXPRESSION 1 THRU EXPRESSION N-1 ARE
.   FALSE AND EXPRESSION N IS TRUE
.
ELSE
.
.   STATEMENTS TO EXECUTE IF EXPRESSION 1 THRU EXPRESSION N ARE FALSE
.
END IF
```

Figure 3.3a. Statement Syntax

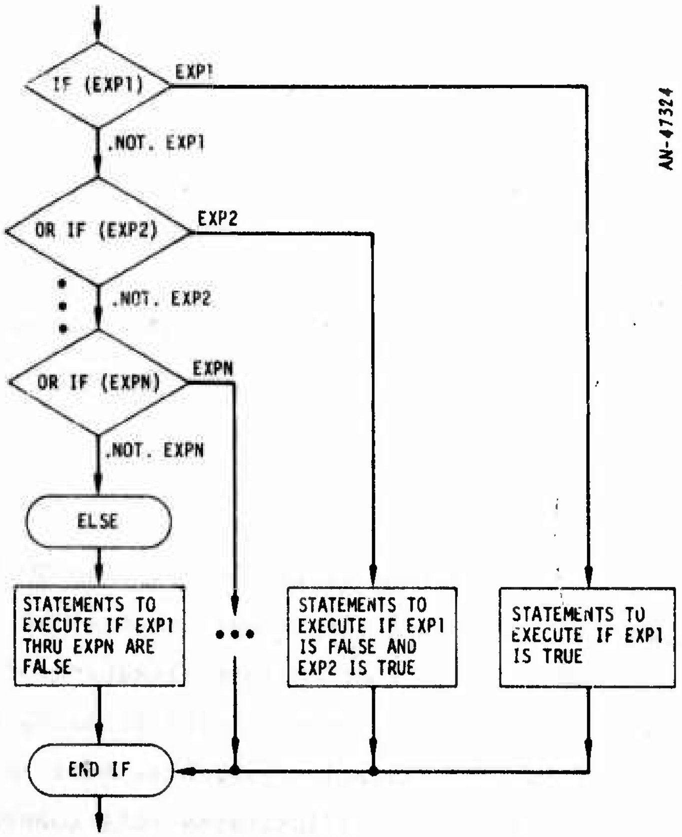

IF (EXP1)
EXP1
.NOT. EXP1
OR IF (EXP2)
EXP2
.NOT. EXP2
OR IF (EXPN)
EXPN
.NOT. EXPN
ELSE
STATEMENTS TO EXECUTE IF EXP1 THRU EXPN ARE FALSE
STATEMENTS TO EXECUTE IF EXP1 IS FALSE AND EXP2 IS TRUE
STATEMENTS TO EXECUTE IF EXP1 IS TRUE
END IF
AN-47324

Figure 3.3b. Construct Flowchart

Figure 3.3. IF...ORIF...ELSE...END IF Construct

WHILE...END WHILE

The WHILE...END WHILE construct indicates a repetitive operation which is to be performed zero or more times. It is essentially a single-exit loop which exits at the top of the loop. Figure 3.4 illustrates the form and meaning of this construct. It is important to note that no initialization or incrementing operations are caused by the WHILE...END WHILE construct. Initialization must be explicitly performed before entering the loop, and the iteration variables must be explicitly modified on each pass through the loop.

DO...END DO

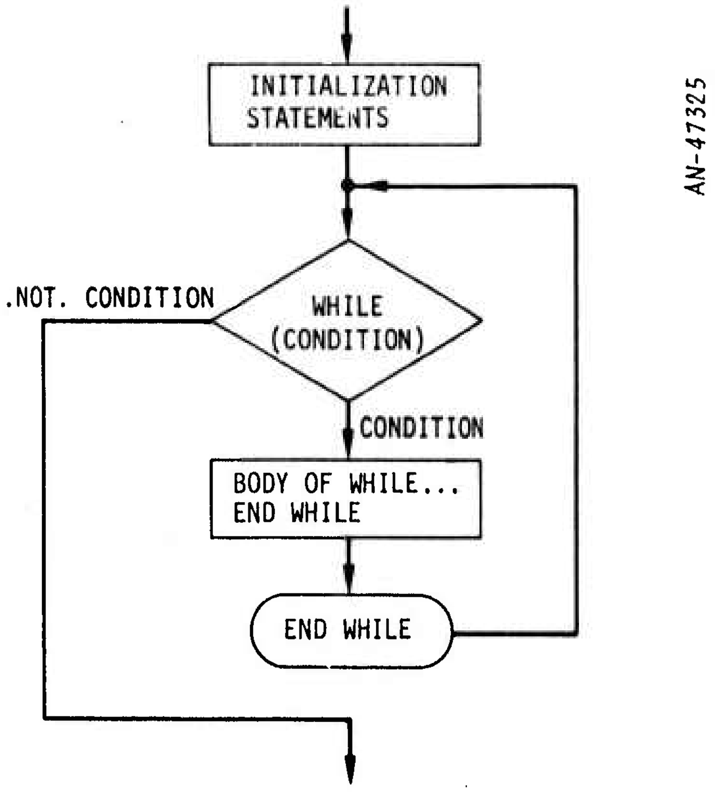
The DO...END DO construct indiciates a repetitive operation that is to be performed one or more times. It is a single-exit loop with the exit at the bottom. Figure 3.5 illustrates this construct. It has the same meaning as the FORTRAN DO-LOOP; however, no label is necessary, and (CHARACTER-STRING) must be of the form (INDEX=INITIAL,FINAL, INCR) where each of these variables is a simple integer variable or constant (except for INDEX, which must be variable). If INCR is not present, it is assumed to be 1. The value of INDEX is not defined after DO...END DO termination. The implied initialization and incrementing operations are indicated in Fig. 3.5b.

REPEAT...UNTIL

The REPEAT...UNTIL construct is like a DO...END DO in that it is performed at least once and has a single exit at the bottom of the loop, and like a WHILE...END WHILE in that no initialization or incrementing operations are caused by this construct. Initialization must be performed before entering the loop, and iteration variables must be modified on each pass through the loop. Figure 3.6 illustrates this construct.

```
INITIALIZATION STATEMENTS
WHILE ( CONDITION )
.
.    BODY OF WHILE...END WHILE
.
END WHILE
```

a) Statement Syntax



b) Construct Flowchart

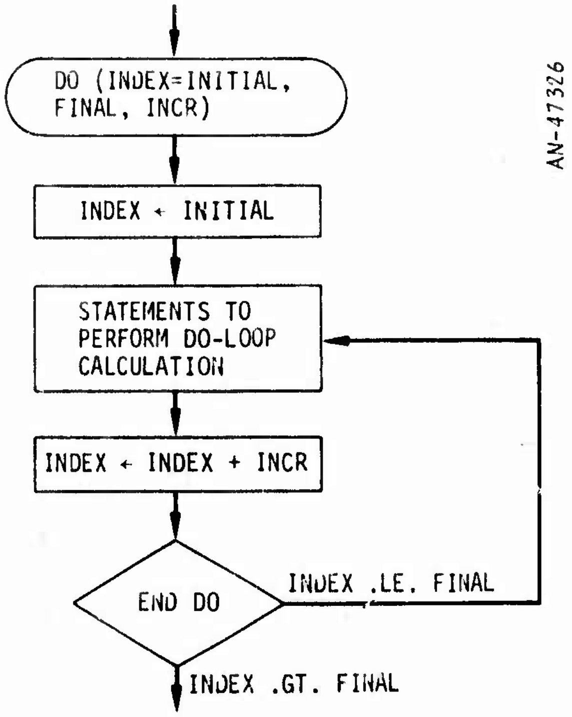
```
FUNCTION SQRT( A )
X = A
WHILE( ABS(X-A/X) .GT. 1.E-6 )
.    X = (X+A/X)/2
END WHILE
SQRT = X
RETURN
END
```

c) FORTRAN Example

Figure 3.4. WHILE...END WHILE Construct

```
DO ( INDEX = INITIAL, FINAL, INCR )
.
.   STATEMENTS TO PERFORM DO-LOOP CALCULATION
.
END DO
```

a) Statement Syntax



b) Construct Flowchart

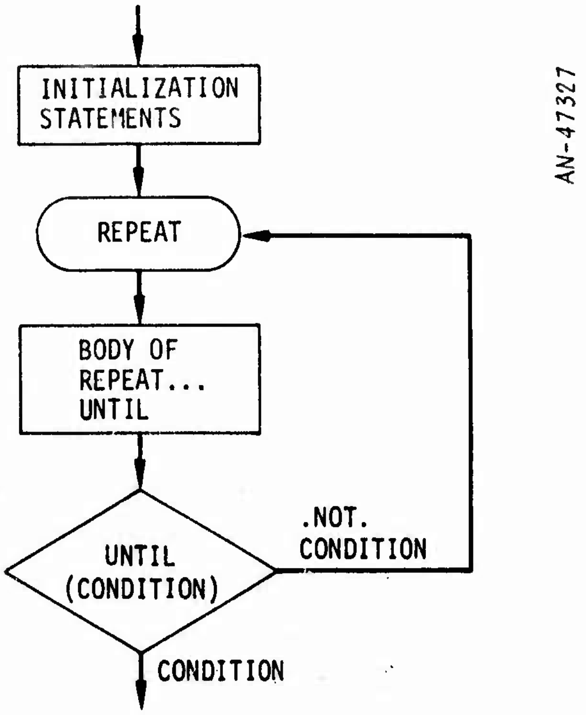
```
SUBROUTINE MLTPLY(A,B,C,N)
DIMENSION A(10,10),B(10,10),C(10,10)
DO(I=1,N)
.   DO(J=1,N)
.   .   S = 0.
.   .   DO(K=1,N)
.   .   .   S = S + A(I,K) * B(K,J)
.   .   END DO
.   .   C(I,J) = S
.   END DO
END DO
RETURN
END
```

c) FORTRAN Example

Figure 3.5. DO...END DO Construct

```
INITIALIZATION STATEMENTS

REPEAT
.
.   BODY OF REPEAT...UNTIL
.
UNTIL ( CONDITION )
```

a) Statement Syntax



b) Construct Flowchart

```
FUNCTION CONVRG(XINIT, EPS, F )
EXTERNAL F
X = XINIT
REPEAT
.    XOLD = X
.    X = F(X)
UNTIL( ABS(X-XOLD) .LE. EPS )
CONVRG = X
RETURN
END
```

c) FORTRAN Example

Figure 3.6. REPEAT...UNTIL Construct

LOOP...EXIT...EXIT IF...END LOOP

The basic LOOP...END LOOP construct is a loop structure with no exit (an infinite loop) and no implied initialization or iteration conditions. At least one EXIT or EXIT IF statement should be used in each LOOP, END LOOP construct. An EXIT or EXIT IF statement is associated with the first END LOOP statement following it. As shown in Figs. 3.7 and 3.8, EXIT and EXIT IF statements cause control to transfer to the first statement after the END LOOP, and are indented to the level of the corresponding LOOP...END LOOP pair. This construct is most useful for loops which naturally exit in the middle or may have more than one reason for exiting.

FOR...END FOR

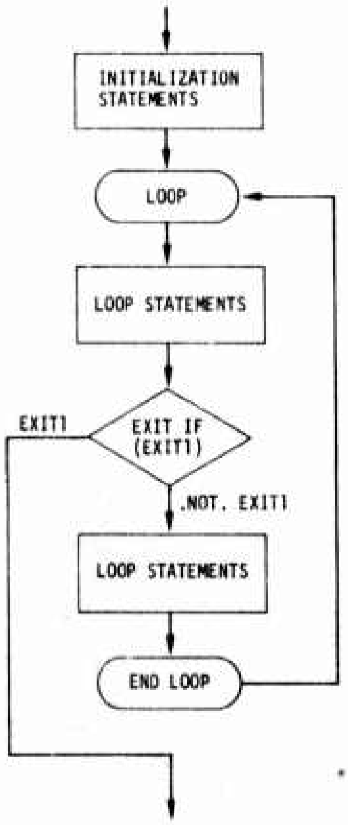
The FOR...END FOR construct allows a variety of loops to be built from an initialization clause, modification clauses, and condition clauses. The initialization clause is required to provide an initial value for the FOR index variable. The index variable will have the last value assigned to it when the FOR...END FOR construct terminates. Optional modification clauses provide for changing the FOR index variable other than incrementing by one (which is the default value). Escape tests at the top or bottom of the FOR loop are constructed with optional condition clauses.

A FOR...END FOR construct with only an initialization clause has the form:

```
FOR (INDEX = INITIAL)
.
.  BODY OF FOR...END FOR
.
END FOR
```

```
INITIALIZATION STATEMENTS

LOOP
.
.   LOOP STATEMENTS
.
EXITIF ( EXIT1 )
.
.   LOOP STATEMENTS
.
END LOOP
```

a) Statement Syntax



AN-47333

b) Construct Flowchart

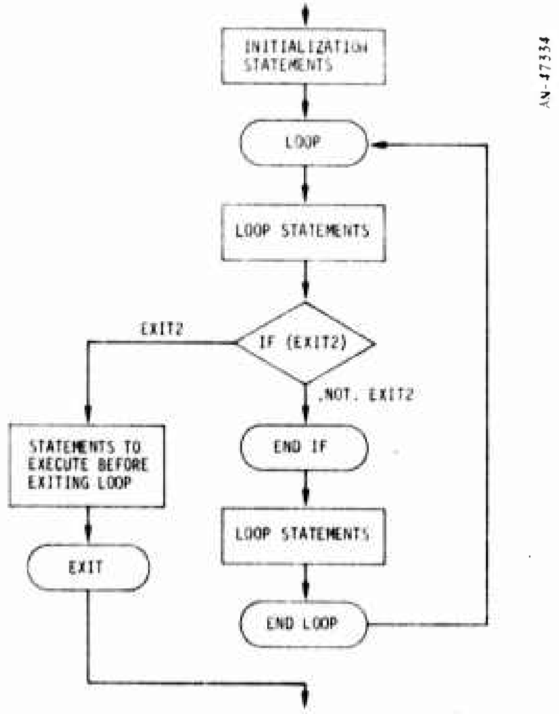
```
   SUBROUTINE COPY( LUNIT )
   INTEGER X(80)
   LOOP
   .   READ(5,10) X
   EXIT IF( EOF(5) )
   .   WRITE( LUNIT, 10) X
   END LOOP
   RETURN
10 FORMAT(80A1)
   END
```

c) FORTRAN Example

Figure 3.7. LOOP...EXIT IF...END LOOP Construct

```
INITIALIZATION STATEMENTS
LOOP
.
.    LOOP STATEMENTS
.
.    IF ( EXIT2 )
.    .
.    .    STATEMENTS TO EXECUTE BEFORE EXITING LOOP
.    .
EXIT
.    .
.    END IF
.
.    LOOP STATEMENTS
.
END LOOP
```

a) Statement Syntax



b) Construct Flowchart

```
      SUBROUTINE INPUT(N,A)
      INTEGER A(1)
      N = 1
      LOOP
      .    READ(5,10) A(N)
      .    IF( EOF(5))
      .    .    N = N - 1
      EXIT
      .    END IF
      .    N = N + 1
      END LOOP
      RETURN
   10 FORMAT(I5)
      END
```

c) FORTRAN Example

Figure 3.8. LOOP...EXIT...END LOOP Construct

INDEX is any variable which is legal on the left-hand side of a FORTRAN assignment statement. The FORTRAN expression "INITIAL" defines the value which is assigned to the index variable when the FOR...END FOR construct is entered. The index variable is incremented by one after the body of the FOR...END FOR is executed. This form will cause an infinite loop. It is shown separately to allow a clearer explanation of the initialization clause alone.

One of the following modification clauses can be used to alter the default incrementing of the index variable. The "BY" clause allows positive or negative increments to be specified. A FOR...END FOR construct with a "BY" clause has the form:

```
FOR (INDEX = INITIAL BY INCR)
.
.  BODY OF FOR...END FOR
.
END FOR
```

A legal FORTRAN arithmetic expression to be used for incrementing is specified with "INCR". This form will also not terminate. The "NEXT" clause can be used to specify new values of the index variable. A FOR...END FOR construct with a "NEXT" clause has the form:

```
FOR (INDEX = INITIAL NEXT EXPR)
.
.  BODY OF FOR...END FOR
.
END FOR
```

"EXPR" is a valid FORTRAN arithmetic expression which is used to compute the next value of the index variable. This form will not terminate.

Several of the following condition clauses can be used to specify FOR loop termination conditions. The "TO" clause is useful to specify a bound on the index variable. This construct has the form:

```
FOR (INDEX = INITIAL TO FINAL)
.
.  BODY OF FOR...END FOR
.
END FOR
```

A "TO" clause and a "BY" clause may be used in the same FOR loop. The "TO" clause alone will cause the index variable to be incremented by one each time through the loop. When the BY clause is used the increment is specified in the clause. A "NEXT" clause cannot be used with a "TO" clause. If the "INCR" specified in the "BY" clause is negative, the appropriate termination test is performed.

The "WHILE" clause allows an escape condition to be inserted at the top of the FOR loop. This construct has the form:

```
FOR (INDEX = INITIAL WHILE COND)
.
.  BODY OF FOR...END FOR
.
END FOR
```

"COND" is a legal FORTRAN logical expression. The index variable will be incremented by one after starting at the INITIAL value. The escape test is at the top of the loop.

The "UNTIL" clause allows an escape condition to be inserted at the bottom of the FOR loop. This construct has the form:

```
FOR (INDEX = INITIAL UNTIL COND)
.
.  BODY OF FOR...END FOR
.
END FOR
```

As in the "WHILE" clause, COND is a legal FORTRAN logical expression. A "WHILE" or "UNTIL" clause can be used with any of the other FOR clauses.

The FOR construct and related clauses can be used to state a wide variety of loops. Restrictions in using the FOR...END construct are:

1. An initialization clause must be used, and it must be the the first clause.
2. The "NEXT" clause cannot be used with either "TO" or "BY" clauses.
3. Each clause keyword (BY, NEXT, TO, WHILE, UNTIL) must be preceded and followed by a blank.

One example of the FOR...END FOR construct is shown in Fig. 3.9a, b, and c.

BLOCK...END BLOCK and INVOKE

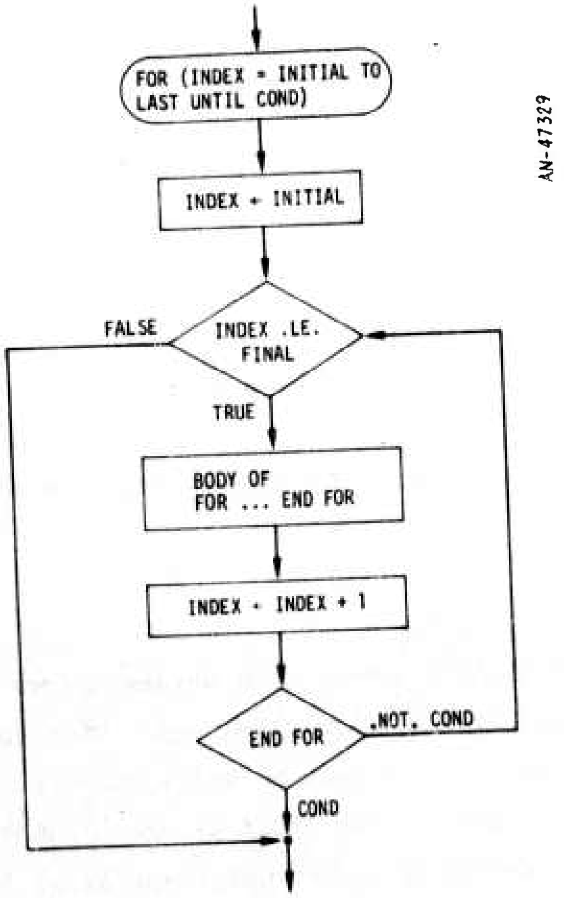
The BLOCK...END BLOCK construct provides a form of internal subroutine capability in IFTRAN source programs. This construct is an internal procedure which has access to all variables in the routine which contains it. A BLOCK...END BLOCK is executed only if it is referred to with an INVOKE statement which specifies its name. The name of the BLOCK below is CHARACTER-STRING:

```
BLOCK(CHARACTER-STRING).
```

All characters in CHARACTER-STRING are significant after the first non-blank and before the last non-blank; (this allows names of more than six characters so that the name can have mnemonic significance). Figure 3.10 illustrates this construct. As the flowchart for this construct indicates, it is a single-entry (the BLOCK statement), single-exit (the END BLOCK statement) section of code. An INVOKE statement causes control to transfer to the named BLOCK statement, and the

```
FOR ( INDEX = INITIAL TO FINAL UNTIL COND )
.
.   BODY OF FOR...END FOR
.
END FOR
```

a) Statement Syntax



b) Construct Flowchart

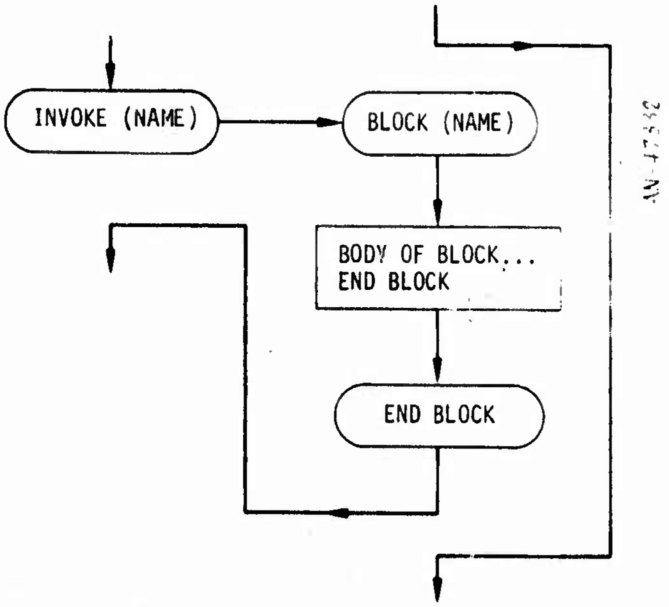
```
INTEGER FUNCTION FSTBLK( LINE )
INTEGER LINE(80)
FOR( I = 1 TO 80 UNTIL LINE(I) .EQ. 1H )
END FOR
IF( I .GT. 80 )
.    FSTBLK = -1
ELSE
.    FSTBLK = I
END IF
RETURN
END
```

c) FORTRAN Example

Figure 3.9. FOR...END FOR With TO and UNTIL Clauses

```
INVOKE ( NAME )

BLOCK (NAME )
.
.   BODY OF BLOCK...END BLOCK
.
END BLOCK
```

a) Statement Syntax



b. Construct Flowchart

```
SUBROUTINE MLTPLY(A,B,C,N)
DIMENSION A(10,10),B(10,10),C(10,10)
DO(I=1,N)
.   DO(J=1,N)
.   .   INVOKE( COMPUTE NEW ARRAY ELEMENT )
.   END DO
END DO

BLOCK( COMPUTE NEW ARRAY ELEMENT )
.   S = 0.
.   DO ( K = 1,N)
.   .   S = S + A(I,K) * B(K,J)
.   END DO
.   C(I,J) = S
END BLOCK
RETURN
END
```

c. FORTRAN Example

Figure 3.10. BLOCK...END BLOCK and INVOKE Construct

matching END BLOCK statement causes control to transfer back to the statement after the INVOKE. More than one INVOKE for a given BLOCK... END BLOCK construct is allowed. Though BLOCK...END BLOCK constructs can be nested, no recursion is allowed in the invoking of BLOCKS (i.e., a BLOCK cannot directly or indirectly invoke itself). Also, the name of a BLOCK is known throughout the entire routine in which it is contained. BLOCKS cannot be invoked from an external routine, nor can they be passed as a parameter to another routine.

3.1.2 PASCAL Control Constructs

The PASCAL language provides a basic framework for programming structured software. Such features as nested procedure declarations, nested control structures, data typing and data structure hierarchy with controlled access by scope are important when writing structured code.

In standard PASCAL, the IF statement, CASE statement, WHILE statement, FOR statement, and WITH statement are all assumed to contain a single statement. In order to allow the inclusion of more than one statement in one of these statements, it is necessary to surround the statements with the words BEGIN and END. The net result is that strings of ENDs can appear in a program which, on occasion, are difficult to pair with their respective BEGINs. One solution has been for a programmer to indent the statements that belong together. Another solution is to improve the syntax so that automatic indenting is possible with a single-pass preprocessor. This relieves the programmer from counting spaces to achieve readability while still providing an indented listing. It also eliminates the need for BEGINs within control structures and assigns more meaning to the ENDs. This is accomplished by using unique keywords to terminate each type of statement or to separate parts of the statement.

Thus, where original PASCAL has

```
IF year > = 500 THEN
     BEGIN write ('d'); year := year - 500 END;
```

the improved syntax lists a statement entered as:

```
IF year > = 500 THEN write ('d'); year := year - 500 END IF;
```

with the indented listing:

```
IF year > = 500 THEN
     write ('d');
     year := year - 500
END IF;
```

In a similar manner, where the original PASCAL has:

```
WHILE power > 0 DO
     BEGIN {ans*temp**power = base**exponent,power > 0}
          WHILE NOT odd (power) DO
               BEGIN power := power DIV 2; temp := sqr(temp)
               END;
          power := power - 1; ans := temp*ans
          END;
```

the improved syntax would list a statement entered as:

```
WHILE power > 0 DO
{ans*temp**power = base **exponent, power > 0}
WHILE NOT odd (power) DO
power := power DIV 2; temp := sqr(temp)
END WHILE;
power := power - 1; ans := temp*ans
END WHILE;
```

with the indented listing:

```
WHILE power > 0 DO
      {ans*temp**power = base **exponent,power > 0}
      WHILE NOT odd (power) DO
            power := power DIV 2;
            temp := sqr(temp)
      END WHILE;
      power := power - 1;
      ans := temp*ans
END WHILE;
```

Where the original PASCAL has a statement of the form:

```
FOR i := 0 TO lim DO
      BEGIN x := d*i; y := exp(-x)*sin(c*x);
      n := round(s*y) + h;
      REPEAT write (' '); n := n - 1
      UNTIL n = 0;
      writeln ('*')
END;
```

The changed syntax provides an indented listing of the form:

```
FOR i := 0 TO lim DO
      x := d*i;
      y := exp(-x)*sin(c*x);
      n := round(s*y) + h;
      REPEAT
            write (' ');
            n := n - 1
      UNTIL n = 0;
      writeln ('*')
END FOR;
```

Where the original PASCAL has a statement:

```
WITH vaccine [child3] DO
      BEGIN month := april; day := 23; year := 1973
      END;
```

the indented PASCAL listing reads:

```
WITH vaccine [child3] DO
      month := april;
      day := 23;
      year := 1973
END WITH;
```

The CASE statement in PASCAL permits the selection of alternative actions based on the evaluation of a single expression. Where the original PASCAL has:

```
CASE i OF
      0: side := 0.;
      1: side := sin(angle);
      2: side := cos(angle);
      3: side := exp(angle);
      4: side := ln(angle)
END
```

The improved syntax looks like this statement:

```
CASE i
OF 0: side = 0.
OF 1: side := sin(angle)
OF 2: side := cos(angle)
OF 3: side := exp(angle)
OF 4: side := ln(angle)
END CASE;
```

and the indented listing:

```
CASE i
      OF 0: side = 0.
      OF 1: side := sin(angle)
      OF 2: side := cos(angle)
      OF 3: side := exp(angle)
      OF 4: side := ln(angle)
END CASE;
```

Note that the separator semicolon (;) has been replaced by the keyword OF.

Other programming languges (e.g., JOVIAL and IFTRAN) offer a language construct for alternative statement selection that is more powerful than the PASCAL CASE statement, namely the IF...ORIF...ELSE construct. The IF, each ORIF, and a terminating ELSE constitute a sequence of alternatives. Each has an associated expression, and the first expression in the sequence which evaluates as true determines the alternative selected. Where in the original PASCAL a succession of nested IF statements is required to state more complex alternatives:

```
IF distance > 0.  THEN answer := distance + minimum
      ELSE IF distance = 0.  THEN answer := abs(minimum)
            ELSE IF distance < minimum THEN answer := minimum
                  ELSE answer := special;
```

the improved syntax reads:

```
IF distance > 0.  THEN answer := distance + minimum
ORIF distance = 0.  THEN answer := abs(minimum)
ORIF distance < minimum  THEN answer := minimum
ELSE answer := special
ENDIF;
```

with the indented listing:

```
IF distance > 0.  THEN
      answer := distance + minimum
ORIF distance = 0.  THEN
      answer := abs(minimum)
ORIF distance < minimum  THEN
      answer := minimum
ELSE
      answer := special
ENDIF;
```

The advantages of these changes are:

1. A readable indented listing is provided whether the programmer indented the source code or not.
2. The END statements do not require comments to tag them to keywords such as END{with}.
3. Numerous BEGINs are eliminated.
4. The flavor of PASCAL is retained.
5. Standard PASCAL can be generated to retain compatibility.

The syntax for these statements is defined in Fig. 3.11; the syntax diagrams are shown in Fig. 3.12.* Figure 3.13 shows the indented listing for a PASCAL program which utilizes most of these control structures.

*The Syntax diagrams shown in Figs. 3.12, 3.15, 3.17, and 3.19 depict changes to those in Ref. 12.

```
*     <if statement>::=

            if <expression> then <statement list> <alternative> end if

      <alternative>:= {orif <expression> then <statement list>}|

            {orif <expression> then <statement list>} else <statement list>

*     <case statement>::=

            case <expression> of <case list element>
            {or <case list element>} end case

*     <case list element>::=<case label list>:<statement list>|

            <empty>

*     <while statement>::=

            while <expression> do <statement list> end while

*     <for statement>::=

            for <control variable>:=<for list> do
            <statement list> end for

*     <with statement>::=

            with <record variable list> do
            <statement list> end with

+     <statement list>::=<statement>{;<statement>}
```

*Denotes change to existing BNF.

+Denotes addition to existing BNF.

Figure 3.11. PASCAL Statements for Control Structures

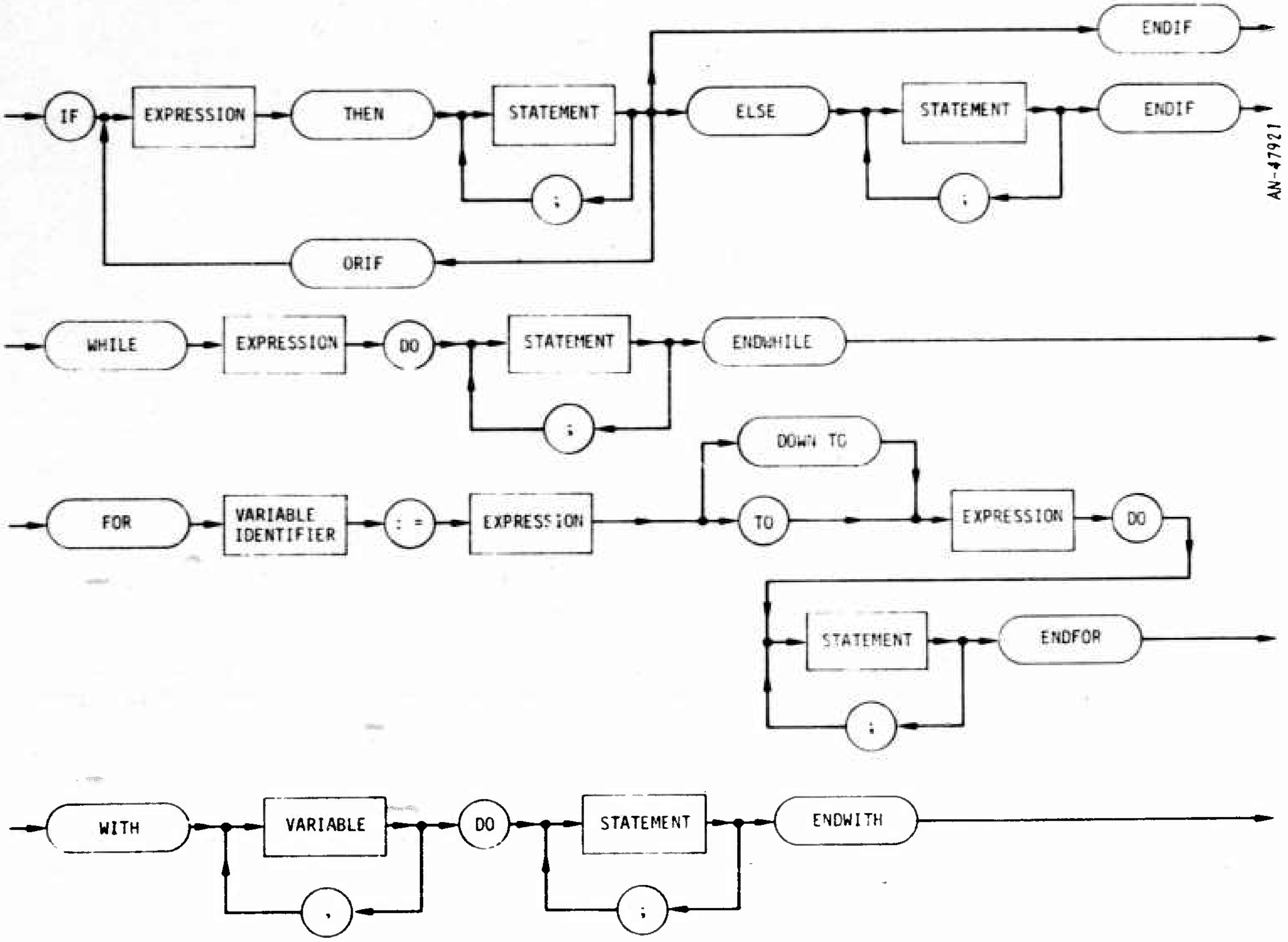


Figure 3.12. Syntax Diagrams for Control Structures

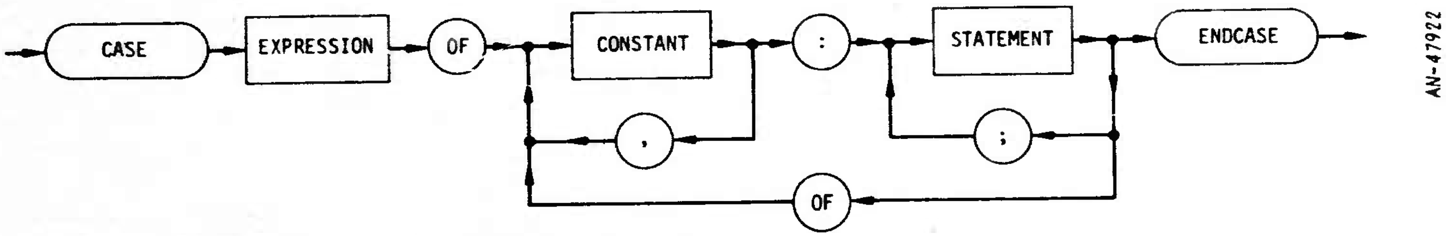


Figure 3.12. Continued

```
 1 H    PROGRAM PRIMES ( OUTPUT ) :                                                            .PRIMES

 2 C    CONST                                                                                  .        2
 3 C         WDLENGTH = 60 : (*IMPLEMENTATION DEPENDENT*)                                      .        3
 4 C         MAXBIT = 58 :                                                                     .        4
 5 C         W = 84 : (*W=N DIV WDLENGTH DIV 2 *)                                              .        5

 6 V    VAR                                                                                    .        6
 7 V         SIEVE , PRIMES : ARRAY [ 0 .. W ] OF SET OF 0 .. MAXBIT :                         .        7
 8 V         NEXT : RECORD                                                                     .        8
 9 V         . WORD , BIT : INTEGER                                                            .        9
10 V         . END :                                                                           .       10
11 V         J , K , I , C : INTEGER :                                                         .       11
12 V         EMPTY : BOOLEAN :                                                                 .       12
                                                                                STATEMENT PART OF PRIMES
13 S  0 BEGIN (*INITIALIZE*)                                                                   .       13
14 S  0      FOR I := 0 TO W DO                                                                .       14
15 S  1      . SIEVE [ I ] := [ 0 .. MAXBIT ] :                                                .       15
16 S  1      . PRIMES [ I ] := [ ]                                                             .       16
17 S  0      ENDFOR :                                                                          .       17
18 S  0      SIEVE [ 0 ] := SIEVE [ 0 ] - [ 0 ] :                                              .       18
19 S  0      NEXT . WORD := 0 :                                                                .       19
20 S  0      NEXT . BIT := 1 :                                                                 .       20
21 S  0      EMPTY := FALSE :                                                                  .       21
22 S  0      WITH NEXT DO                                                                      .       22
23 S  1      . REPEAT (*FIND NEXT PRIME*)                                                      .       23
24 S  2      . . WHILE NOT ( BIT IN SIEVE [ WORD ] ) DO                                        .       24
25 S  3      . . . BIT := SUCC ( BIT ) :                                                       .       25
26 S  2      . . ENDWHILE :                                                                    .       26
27 S  2      . . PRIMES [ WORD ] := PRIMES [ WORD ] + [ BIT ] :                                .       27
28 S  2      . . C := 2 * BIT + 1 :                                                            .       28
29 S  2      . . J := BIT :                                                                    .       29
30 S  2      . . K := WORD :                                                                   .       30
31 S  2      . . WHILE K <= W DO (*ELIMINATE*)                                                 .       31
32 S  3      . . . SIEVE [ K ] := SIEVE [ K ] - [ J ] :                                        .       32
33 S  3      . . . K := K + WORD * 2 :                                                         .       33
34 S  3      . . . J := J + C :                                                                .       34
35 S  3      . . . WHILE J > MAXBIT DO                                                         .       35
36 S  4      . . . . K := K + 1 :                                                              .       36
37 S  4      . . . . J := J - WDLENGTH                                                         .       37
38 S  3      . . . ENDWHILE                                                                    .       38
39 S  2      . . ENDWHILE :                                                                    .       39
40 S  2      . . IF SIEVE [ WORD ] = [ ] THEN                                                  .       40
41 S  3      . . . EMPTY := TRUE :                                                             .       41
42 S  3      . . . BIT := 0 :                                                                  .       42
43 S  2      . . END IF :                                                                      .       43
44 S  2      . . WHILE EMPTY AND ( WORD < W ) DO                                               .       44
45 S  3      . . . WORD := WORD + 1 :                                                          .       45
46 S  3      . . . EMPTY := SIEVE [ WORD ] = [ ] :                                             .       46
47 S  2      . . ENDWHILE :                                                                    .       47
48 S  1      . UNTIL EMPTY : (*ENDS WITH*)                                                     .       48
49 S  0      END WITH :                                                                        .       49
50 S  0 END .                                                                                  .       50
```

Figure 3.13. Example Showing Improved PASCAL Control Structures

3.2 EXECUTABLE ASSERTIONS

The addition of assertions to a language provides the designer and programmer with a tool for stating specifications that can be used as a basis for static consistency checking and program verification as well as for execution tests. This capability has been added to FORTRAN and PASCAL without disrupting the integrity of the standard languages. The existing definition of expressions was retained, and the ability to state quantified logical expressions was added.

Three keywords: INITIAL, ASSERT, and FINAL were added to the language to allow the expression of assertions. As the names imply, INITIAL is used for an assertion that is initially true on entry to a program or procedure; FINAL is used for a final assertion that is true on exit from a program or procedure; and ASSERT is used anywhere else.

The syntax of the INITIAL/ASSERT/FINAL assertions is:

FORTRAN	PASCAL
KEYWORD (assertion expression)	KEYWORD assertion expression

where KEYWORD is INITIAL, ASSERT, or FINAL and is followed by any expression that evaluates to true or false. Examples are

FORTRAN

```
ASSERT (LINEND .GT. ZERO)
FINAL (VOLUME .EQ. HEIGHT*AREA)
INITIAL (COLOR .EQ. RED .AND. SIZE .EQ. 10)
```

PASCAL

```
ASSERT linend > zero;
FINAL volume = height*area;
INITIAL (color = red) AND (size = 10);
```

One logical operator was added to the expression used in assertion statements. This is the implication operator which is represented by .IMP. in FORTRAN and by the pair of symbols => in PASCAL. Thus it is possible to state

```
FORTRAN                       PASCAL
ASSERT (P .IMP. Q)            ASSERT p => q;
ASSERT (A .AND. B .IMP. Q)    ASSERT a AND b => q;
```

The implication operator has the lowest precedence of all operators which means it is the last operator to be evaluated.

Quantified expressions are defined in terms of the keywords SOME which stands for $\exists$ and ALL which stands for $\forall$. A range of values is stated for the quantified variable in a similar manner to PASCAL FOR statements. Examples of assertions which contain quantified expressions are:

```
FORTRAN
      ASSERT (.ALL. I IN (FIRST, LAST) (ARRAY(I)
            .LT. ARRAY (I - 1)))
      INITIAL (.SOME. J IN (1, K) (ARRAY(J)
            .EQ. ANSWER))

PASCAL
      ASSERT ALL (i IN 1 TO n IS
            a[i] < a[i + 1])
      INITIAL SOME (j IN 1 TO k IS
            x[j] = answer)
      FINAL ALL (m in 1000 DOWN TO 1 IS
            y[m] >= x[n])
```

With these changes, the full range of expressions in the first-order logic may be stated. However, there is one more type of expression that is useful in assertion statements. This is the definition of an assertion that can be used as a function. For example, to assert that an array subtotal has been zeroed requires the statement

```
ASSERT ALL (i IN 1 TO length IS
      subtotal[I] = 0);
```

However, if this type of assertion is common, a function named zeroed can be defined so that we can state

```
ASSERT zeroed (subtotal);
```

which requires fewer symbols and is more mnemonic. The declaration of this function is a Boolean type PASCAL function whose body contains only assertion statements.

```
PROCEDURE zeroed (a: ARRAY [1..length] OF integer): boolean;
      VAR i: integer;
      BEGIN
            ASSERT ALL (i IN 1 TO length IS
                  a[i] = 0)
      END; {zeroed}
```

Upon exit from the procedure, the value of the function is set to the conjunction of all assertions.

The corresponding assertion form which would be recognized by the IFTRAN preprocessor is

```
ASSERT (ZEROED (SBTOTL,LENGTH))
```

The code for the function would be

```
      LOGICAL FUNCTION ZEROED (SBTOTL,LENGTH)
CMODN ZEROED
      LOGICAL ASSERT
      ASSERT (.ALL. I IN (1, LENGTH) (A(I) .EQ. 0))
      ZEROED = ASSERT
      RETURN
      END
```

The assertion statement may include a FAIL clause. The FAIL construct is a vehicle for defining an exception action which is executed whenever the assertion expression evaluates to false. In an IFTRAN program, this is accomplished with a FAIL statement which invokes an error-processing routine specified by the user and contained within a BLOCK...END BLOCK construct. Following execution of the BLOCK procedure, control is transferred back to the statement following the FAIL. The syntax of the FAIL construct is

```
    KEYWORD (assertion expression)
    FAIL (name of BLOCK)
    .
    .
    .
    BLOCK (name of BLOCK)

          .
          .
          .
    END BLOCK
```

where KEYWORD is INITIAL, ASSERT, or FINAL. An example is

```
    INITIAL ( INDEX .GT. 0 )
    FAIL ( PRINT ERROR CAUSE )

    .
    .
    BLOCK ( PRINT ERROR CAUSE )
    WRITE ( LOUT,1)
  1 FORMAT ( 41HOINPUT PARAMETER HAD NO MEANINGFUL VALUE )
    END BLOCK
```

In PASCAL, a sequence of statements for the exception action starts with the keyword FAIL and terminates with an END FAIL. The exception action can be used to recover from erroneous data as in the example:

```
    ASSERT ALL (i IN 1 TO 20 IS z > a[i])
          FAIL recover(z) END FAIL;
```

3.2.1 FORTRAN

INITIAL, FINAL, and ASSERT statements are normally changed to FORTRAN comments by the IFTRAN preprocessor, but may optionally be changed to execute FORTRAN statements which provide exception reports whenever the assertion is not true during execution. The translation templates are presented in Appendix C.

Assertions for ASSERT may be placed anywhere a statement may be used. The INITIAL and FINAL assertions are placed, respectively, immediately before and after the executable code. It is also possible to include any or all of these assertions within a BLOCK..END BLOCK construct; the same rules for placement apply.

3.2.2 PASCAL

The PASCAL assertion for INITIAL is placed immediately after the BEGIN which starts the program or procedure statements (or after a label referenced by non-local GOTO statement*), and the assertion for FINAL is placed immediately before the END which terminates the statement part of a program or procedure (or before a GOTO which transfers control to a non-local label**). Assertions for ASSERT may be placed anywhere a statement may be used. The syntax for executable assertions is shown in Fig. 3.14 and the syntax diagrams are in Fig. 3.15. The translation templates are in Appendix B.

3.3 DATA ACCESS STATEMENTS

In standard PASCAL and FORTRAN, a program module has access not only to locally declared variables but also to variables which are declared outside the scope of the module. Access to global variables occurs in one of two ways: by explicit declaration (i.e., those appearing as formal parameters or, in FORTRAN, as common variables) and, in

* This construct violates the single <u>entry</u> restriction for a module.

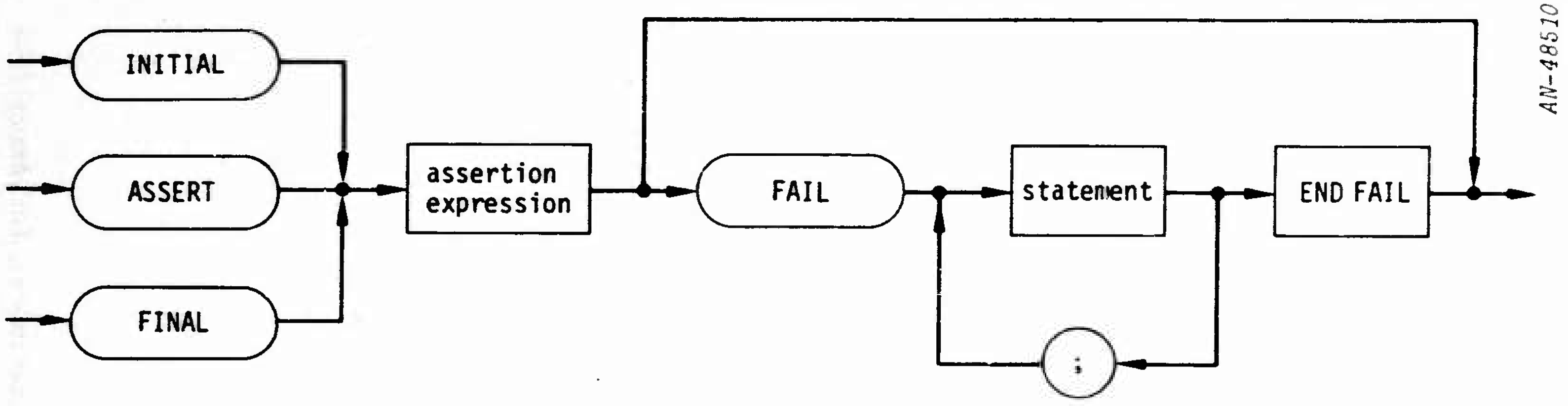
** This construct violates the single <u>exit</u> restriction for a module.

```
*    <unlabelled statement>::=
          <simple statement>|<structured statement>|
          <assertion>

+    <assertion>::= <initial assertion>|<assertion statement>|<final assertion>

+    <initial assertion>::=
          initial <assertion expression>|
          initial <assertion expression> fail <statement list> end fail


+    <final assertion>::=
          final <assertion expression>
          final <assertion expression> fail <statement list> end fail


+    <assertion statement>::=
          assert <assertion expression>;|
          assert <assertion expression> fail <statement list> end fail

+    <assertion expression>::=
          <quantified expression>|<quantified expression>
          => <quantified expression>

+    <quantified expression>::=
          all <quantifier tail>| some <quantifier tail>|
          <expression>

+    <quantifier tail>::=
          (<control variable> in <quantifier list> is
          <assertion expression>)

+    <quantifier list>::=
          <initial value> to <final value>|
          <initial value> down to <final value>

+    <initial value>::= <expression>

+    <final value>::= <expression>
```

*Denotes change to existing BNF

+Denotes addition to existing BNF

Figure 3.14. PASCAL Statements for Assertions



additional lines in statement graph

Figure 3.15. Syntax Diagram for PASCAL Assertions

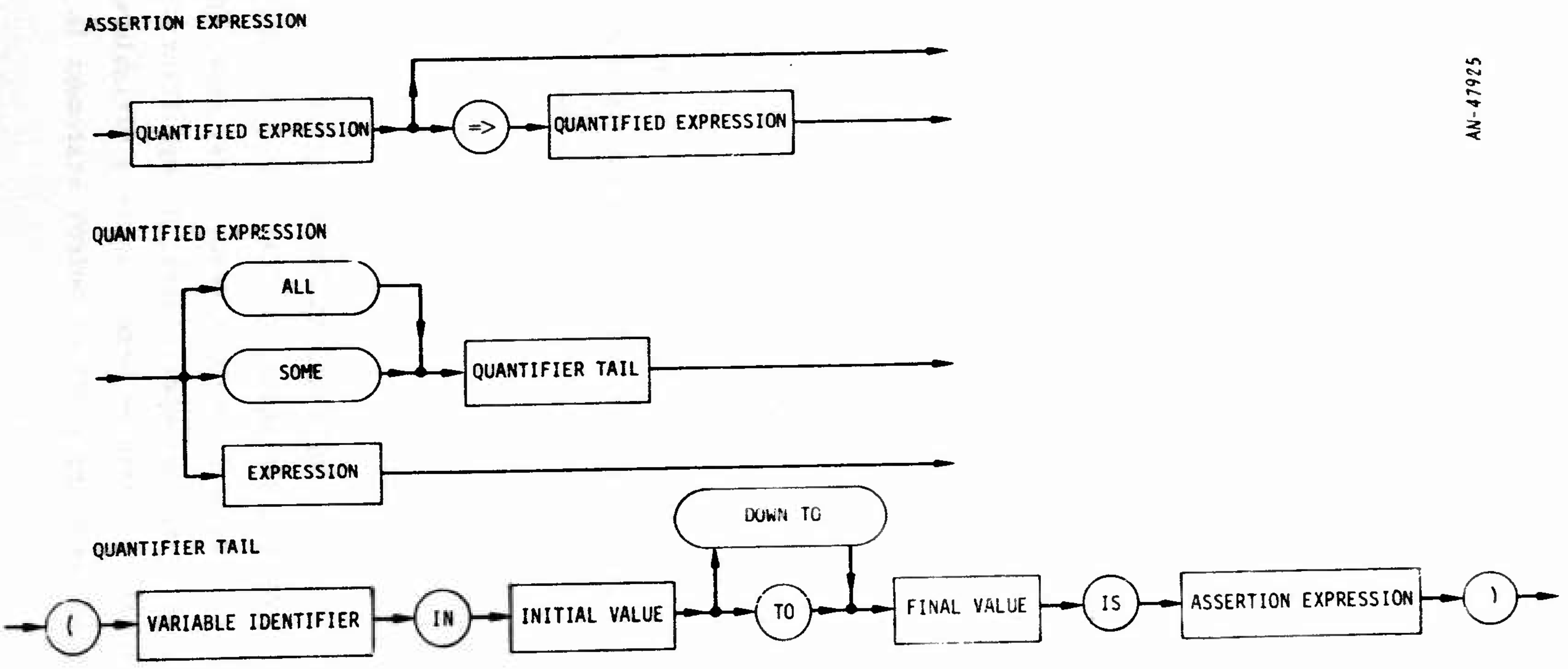


Figure 3.15. Continued

standard PASCAL, by implicit scope rules (i.e., all variables declared in the set of other modules which contain the particular module).

INPUT and OUTPUT assertions specify the data access rights to global variables and are used in static consistency checking as well as dynamic testing. The INPUT statement lists those variables which are input (i.e., set prior to entry) to the module. The OUTPUT statement lists those variables which are output (e.g., assigned or read from auxiliary storage) by the module. This restricts the global variables which may be used or set by the particular module.

3.3.1 FORTRAN

INPUT and OUTPUT statements are normally changed to FORTRAN comments by the IFTRAN preprocessor, but may optionally be changed to executable FORTRAN statements which cause the values of program variables to be printed. If the source language embedded in the IFTRAN program is FORTRAN, and dynamic tracing of input or output variables will be performed, a specific syntax for INPUT and OUTPUT statements is required. In order to print with a correct format, the INPUT and OUTPUT statements must provide type specifications. Any variables whose type is not specified will not be printed. The syntax to provide type information is

```
INPUT (VARIABLE-LIST1)
```

or

```
OUTPUT (VARIABLE-LIST2)
```

where each VARIABLE-LIST is /TYPE/VARIABLE1,/TYPE/VARIABLE2, ... and TYPE is one of REAL, INTEGER, HOLLERITH, LOGICAL, or none. Each VARIABLE may be a non-subscripted variable name, array name, individual element of an array, or array subrange. A(I=1, N) specifies a subrange where A is an array of N words or more. I is a variable whose value will be undefined after the INPUT or OUTPUT statement is executed.

Type specifications preceding each VARIABLE are optional. A type specification remains in effect until it is changed. Only variables with REAL, INTEGER, HOLLERITH, or LOGICAL type specifications will be printed. The VARIABLE-LIST

```
INDEX,/REAL/RANGE,DIST,/INTEGER/L(I = M,N)
```

will not print a value for INDEX, but will print the values for RANGE and DIST with an E or F format and the Mth to Nth values of array L with an integer format. The variables of VARIABLE-LIST1 and VARIABLE-LIST2 need not be distinct; that is, a variable may be used as both input and output. Figure 3.1 provides a specific example of INPUT and OUTPUT statements.

The INPUT statement should immediately precede the first executable statement of a program or block. The OUTPUT statement should precede the RETURN or STOP statement at the end of the program or an imbedded RETURN or STOP which is before the end of the program. If used within a BLOCK, it is placed after all executable code but before the END BLOCK.

3.3.2 PASCAL

The INPUT statement is positioned after the first BEGIN in the body of the module. Labels referenced by non-local GOTO statements should be followed by an INPUT statement. The OUTPUT statement is positioned prior to the last END in the module body. GOTO statements which reference non-local statement labels should be immediately preceded by an OUTPUT statement. The syntax of INPUT and OUTPUT statements is shown in Fig. 3.16 and the syntax diagrams in Fig. 3.17

3.4 PHYSICAL UNITS STATEMENTS

The extensions to FORTRAN and PASCAL include a specification capability for the physical units with each constant or variable. This permits automated units consistency checking in expressions during static analysis. A UNITS statement performs no executable function in

<input statement>::= input <variable identifier>{,<variable identifier>}

<output statement>::= output <variable identifier>{,<variable identifier>}

Figure 3.16. Data Access Statements for PASCAL

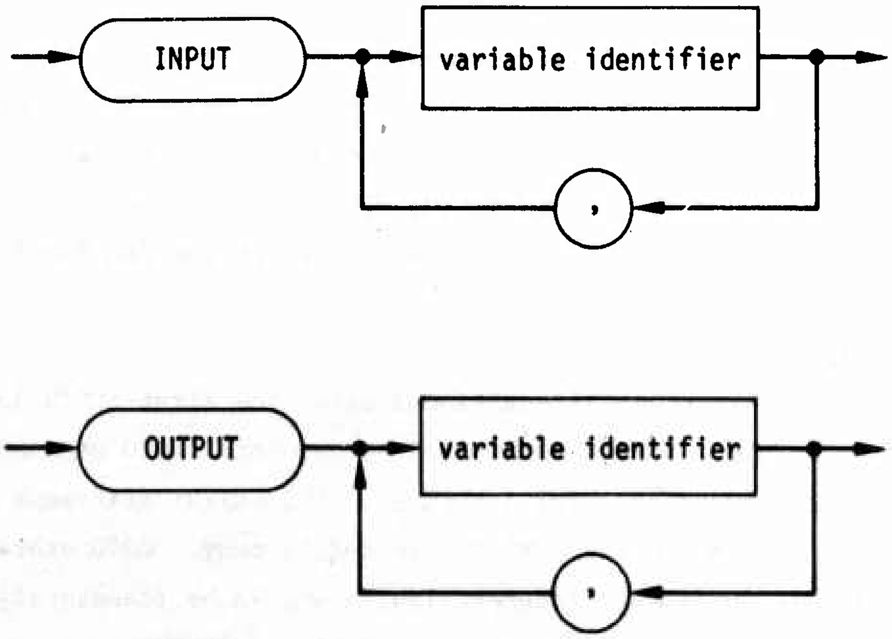


Figure 3.17. Syntax Diagrams for PASCAL Data Access Statements

a dynamic test and is changed to a comment. Physical units are expressed as a quotient of products (e.g., FT/SEC, LB*SEC**2, 1/GRAM) for actual units, or the symbol 1 for unitless, or the symbol 0 for arbitrary units.

3.4.1 FORTRAN

The syntax for the UNITS statement is

```
UNITS(VARIABLE-LIST1 = UNITS-EXPRESSION1,
          VARIABLE-LIST2 = UNITS-EXPRESSION2,...)
```

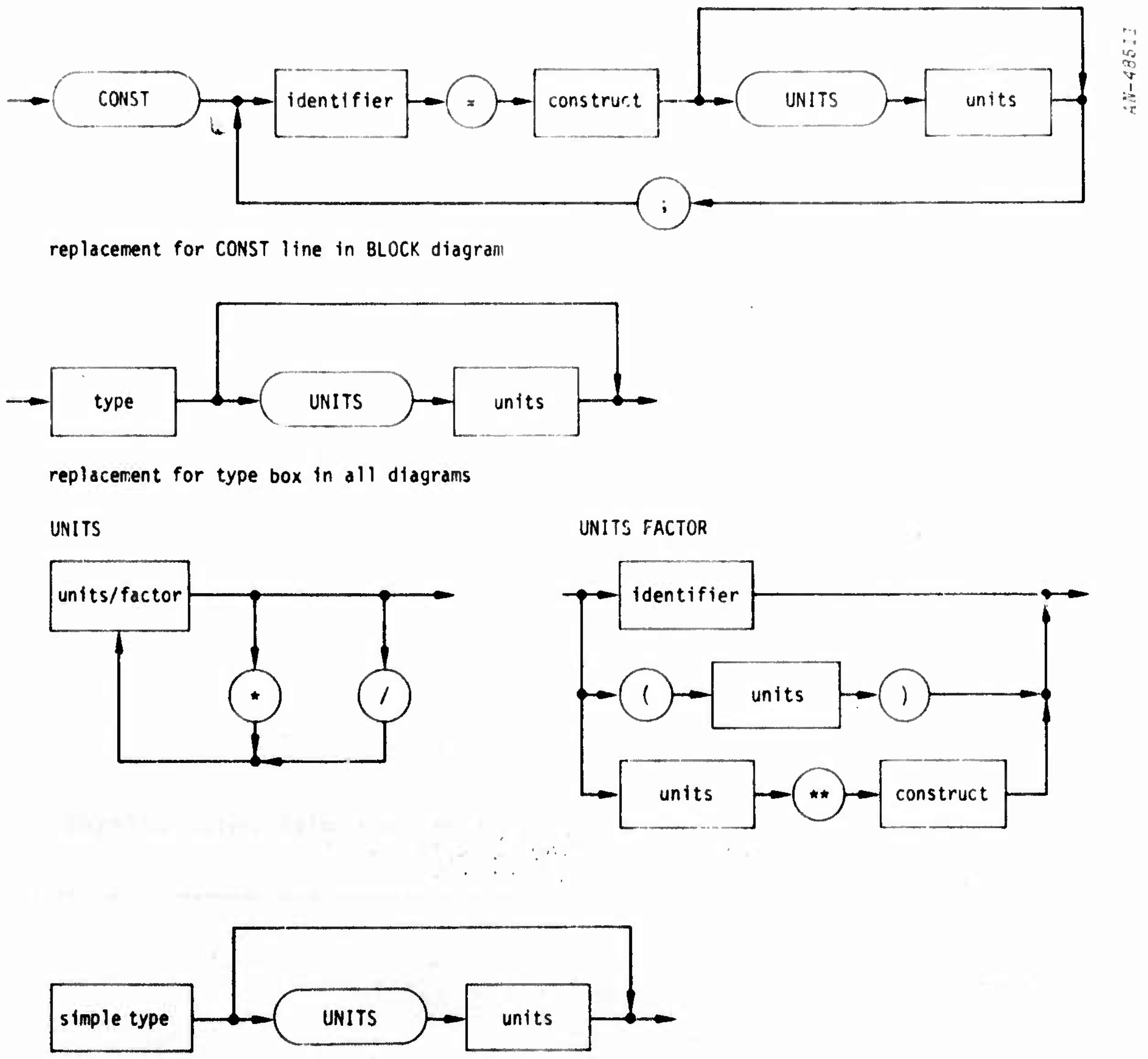
where each VARIABLE-LIST is VARIABLE1 = VARIABLE2 = ... and each UNITS-EXPRESSION is an arithmetic expression involving the physical units of the program variables. All UNITS statements should be included in the declaration section before any executable statements. The second statement in Fig. 3.1 provides an example of a UNITS statement.

3.4.2 PASCAL

Standard PASCAL has provision for assigning types to constants and variables. In addition hierarchies of data structures may be constructed through type declarations. These represent the computer implementation of the program data as opposed to a physical interpretation (e.g., physical units) of the data. Some PASCAL compilers strongly enforce data types and data usage (i.e., only permitted operations are allowed on a specific data type and so-called "mixed mode" expressions are handled with type conversion rules or are considered to be errors). For PASCAL, the syntax of units declarations is a qualifier to the simple type or to the constant definition. The syntax for the UNITS qualifier is shown in Fig. 3.18, and the syntax diagram is in Fig. 3.19. Figures 3.20 and 3.21 show examples of usage.

```
*     <constant definition>::=
            <identifier> = <constant>|
            <identifier> = <constant> units <units>

*     <simple type>::= <basic type>|<basic type> units <units>
+     <basic type>::= <scalar type>|<subrange type>|<type identifier>

+     <units>::
            <units factor>|<units>/<units factor>|
            <units>*<units factor>

+     <units factor>::=
            <identifier>|(<units>)|<units>**<constant>
```

Figure 3.18. UNITS Qualification for PASCAL



replacement for CONST line in BLOCK diagram

replacement for type box in all diagrams

UNITS

UNITS FACTOR

replacement for simple type in all diagrams

Figure 3.19. Syntax Diagram for PASCAL UNITs Statements

```
program convert(output);
{generates a table to convert between degrees centigrade
and degrees fahrenheit}

const
      addin = 32 units degfahrenheit;
      mulby = 1.8 units degfahrenheit/degcentigrade;
      low   = 0 units degcentigrade;
      high  = 39 units degcentigrade;
      separator = '------------';
var
      degree:low..high;
begin
      writeln(separator);
      for degree := low to high do
            write(degree,'c',round(degree*mulby+addin),'f');
            if odd (degree) then
                  writeln
            endif
      end for;
      writeln;
      writeln(separator)
end.
```

Figure 3.20. Example of Constant Definition Part with Units Defined

```
type alpha = packed array [1:10] of char;
            payroll = record
                  name: record first, last:alpha
                        end record;
                  SS : integer;
                  time worked: real units hours;
                  rate: real units dollars/hour;
                  pay: real units dollars;
                  end record;
```

Figure 3.21. Example of Type Definition Part with Units Defined

3.5 IMPLEMENTATION OF VERIFIABLE PASCAL

The ability to verify and execute Verifiable PASCAL programs is possible by the addition of a PASCAL front end to the verification condition generator and by a Verifiable PASCAL preprocessor which translates the control structures and assertions into Standard PASCAL. Since PASCAL has been selected as the base language for future Advanced Quality Assurance research projects, extensions and improvements to the language other than those described in Secs. 3.1-3.4 are expected (e.g., constructs for fault tolerant software and for concurrency). This means that the Verifiable PASCAL should be easily adapted for language changes.

Several approaches to developing the PASCAL processor were considered:

1. Modify existing language tools (e.g., the PDL-1 translator or the IFTRAN translator) to analyze PASCAL
2. Design and develop a new processor based on the properties of PASCAL
3. Use a compiler writing system such as CWIC[14] or CWS[15] to generate the PASCAL processor

The existing language analyzer tools are, for the most part, based on FORTRAN languages and extensions such as IFTRAN. The PASCAL language with its extensions is richer than FORTRAN in the very features most important to Advanced Quality Assurance research (i.e., data structure and control structure), thereby adding complexity to the requirement for the language analyzer. The existing front end tools for the verification condition generator are separate and distinct from the translator preprocessor. Because of the complexities of PASCAL, and the requirement for analyzing future language expressions, a single language analyzer to serve both functions (i.e., as front end to the verification condition generator and as a preprocessor for the PASCAL compiler) was desirable. This approach has the advantage of maintaining compatibility in source recognition between the two functions, an improvement over existing tools.

Previous experience in using CWIC[14] and CWS[15] for other applications[16] demonstrated the effectiveness, flexibility, and efficiency of using a compiler writing system to implement language analysis tools over conventional implementation techniques. CWIC offers an extensive library of semantic actions to support syntactic analysis, but presents serious integration problems with other Software Quality Laboratory tools. CWS, on the other hand, is itself written in PASCAL, is moderate in size, and generates a language analyzer written in PASCAL. The resulting program is easier to interface with other tools. Using CWS also offered an opportunity to gain experience with operational PASCAL programs.

CWS is available in two versions: the original bottom-up system[17] used by TRW in RSL development[18] and the later top-down system.[19] GRC had previously obtained both systems, conducted informal experiments using the grammar of PASCAL, and found the top-down system to be more applicable and less restrictive for analyzing large grammars.

The top-down version of CWS consists of a sequence of three programs. The first program, ANALGEN, accepts as input a syntax definition of the language grammar with imbedded semantic actions, analyzes the grammar for completeness and conformance to the properties of LL(1) grammars, prepares tables for the later phase of the system, and generates the procedures which perform the syntactic analysis and semantic analysis of source text. The second program, SCANGEN, using the tables produced by ANALGEN, together with a file of prototype code, constructs the lexical analyzer with its required tables for reserved words and operators; it also incorporates user-specified options for such details as treatment of blanks, length, and maximum number of identifiers and other token-related definitions. The third program, ANDEGEN, completes the language analyzer in preparation for compilation. It merges code generated by both preceding phases together with user-supplied supplementary declarations supporting semantic actions to produce the language analyzer program.

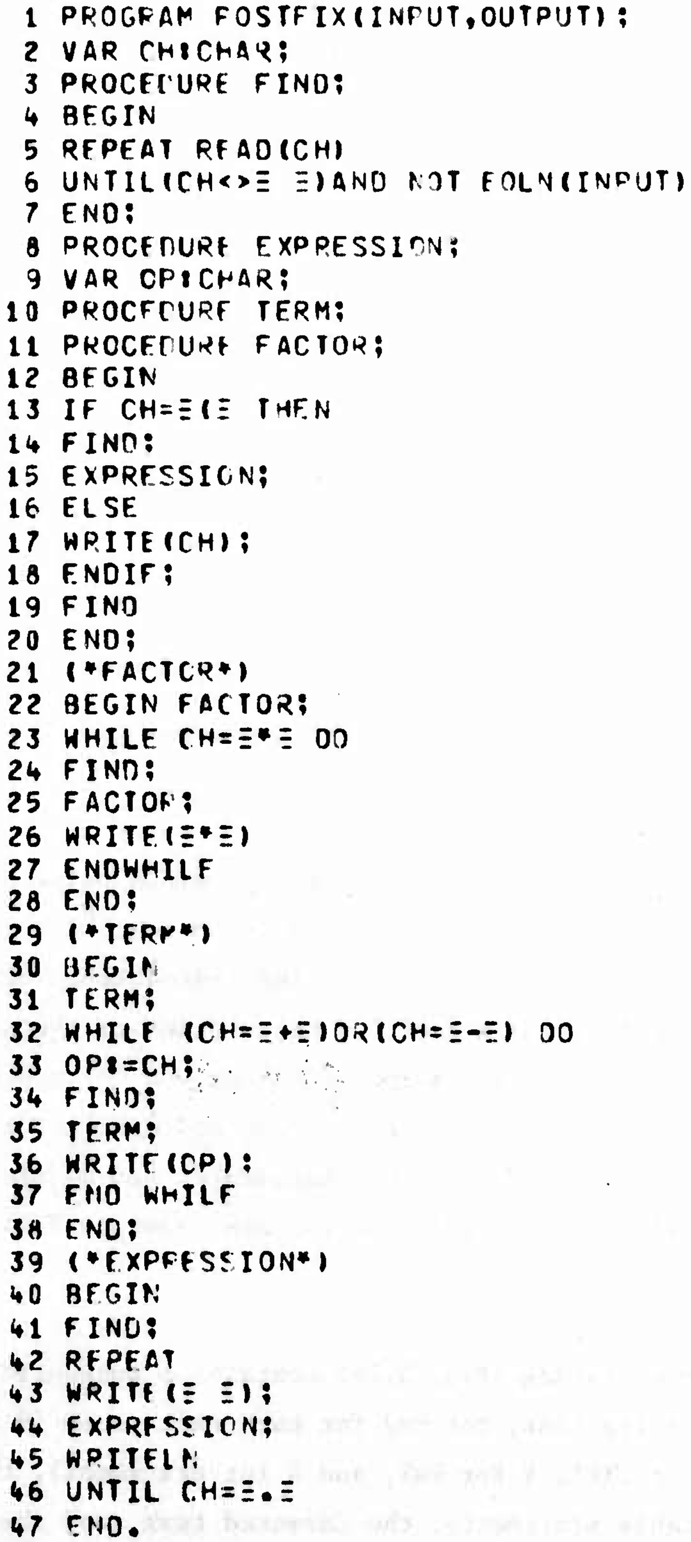
Within the CWS framework a Verifiable PASCAL analyzer has been designed and partially implemented. This analyzer serves both as a front end to the VCG and other tools (incomplete) and as a translator to standard PASCAL (completed). This effort involves the following activities:

1. Develop a description of the grammar for Verifiable PASCAL (see Appendix A).
2. Design standard PASCAL templates for translation of language extensions (see Appendix B).
3. Design and implement semantic actions for language translation
4. Design the interface to the VCG and other tools
5. Design and implement semantic actions for interface

Items 1 through 3 are completed and work has commenced on the remaining two.

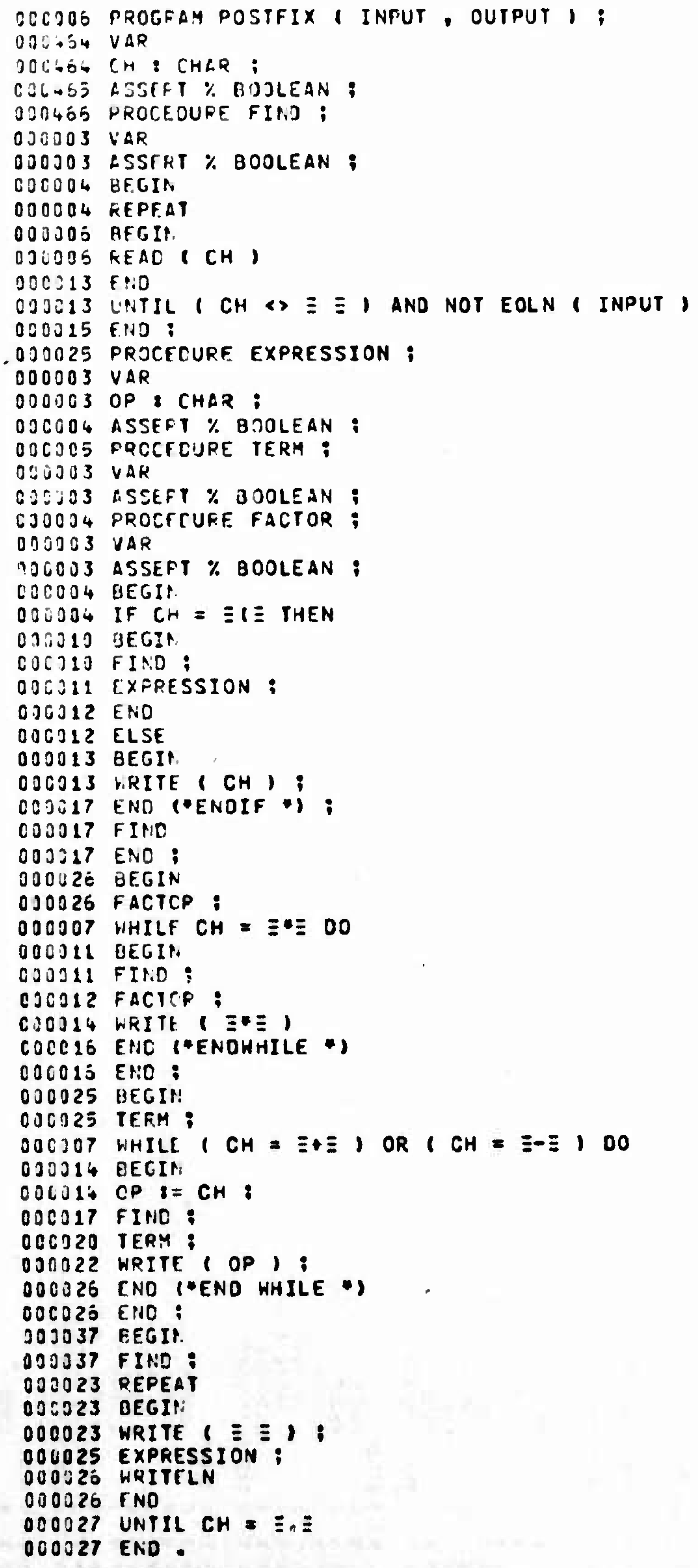
The Verifiable PASCAL analyzer reads programs written in Verifiable PASCAL and generates (1) a listing of the input text with imbedded syntactic error messages, (2) a file of the translated program for input to the PASCAL compiler, (3) a detailed description of the source text suitable for input to the VCG and other tools, and (4) a series of reports including an indented listing of source text and a directory showing text groups (e.g., CONST, TYPE, VAR, statements) and module block structure. Examples of (1), (2), and (4) are shown in Figs. 3.22 through 3.25.

The indented listing (Fig. 3.24) contains a sequential statement number for the entire text, the key for each text group (H for heading, C for CONST, T for TYPE, V for VAR, and S for statement), the nesting level for executable statements, the indented text, and the module block structure with sequential statement number within the module. Assertion warnings in the executed program list the appropriate statement numbers



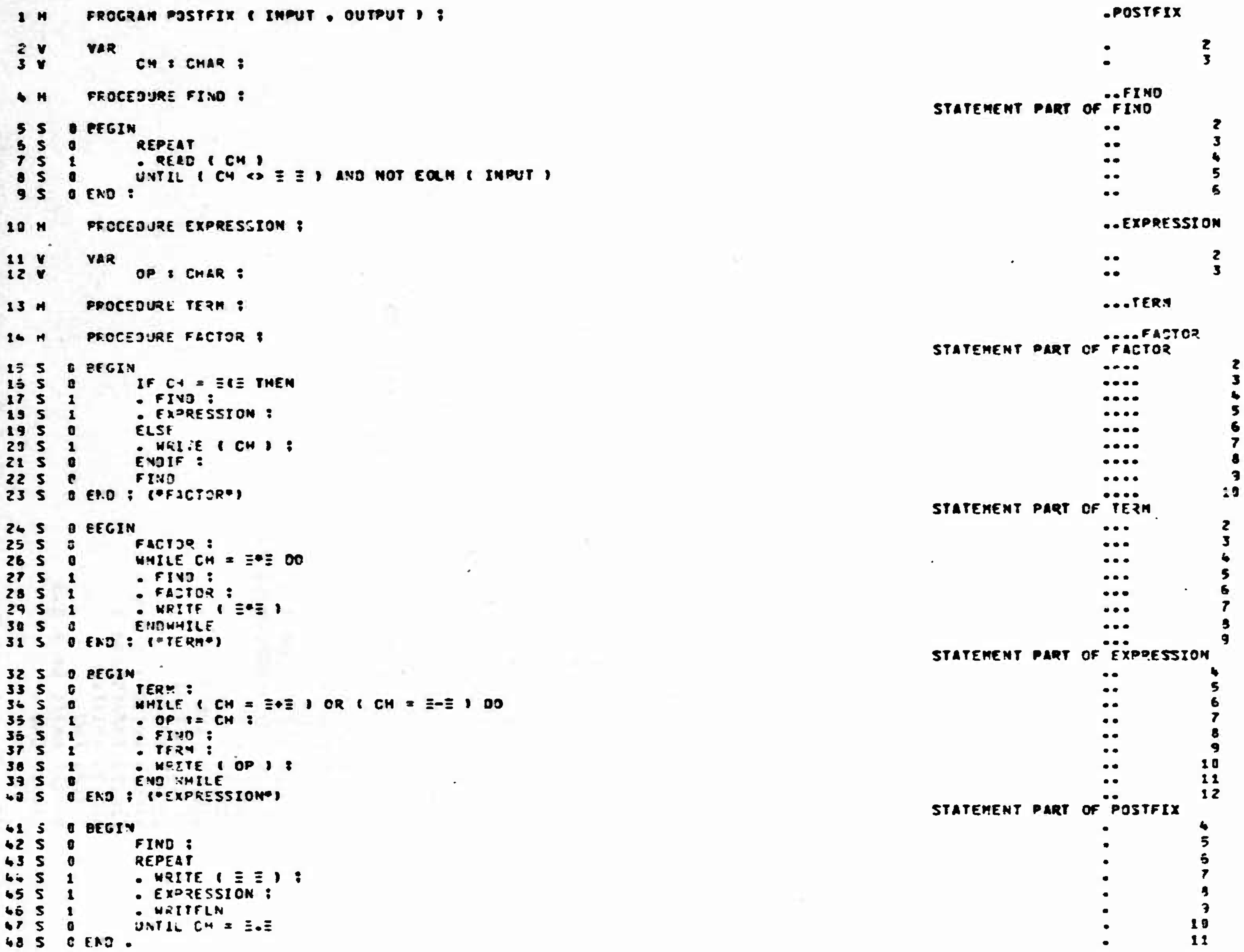
```
 1 PROGRAM FOSTFIX(INPUT,OUTPUT);
 2 VAR CH:CHAR;
 3 PROCEDURE FIND;
 4 BEGIN
 5 REPEAT READ(CH)
 6 UNTIL(CH<>Ξ Ξ)AND NOT EOLN(INPUT)
 7 END:
 8 PROCEDURE EXPRESSION;
 9 VAR OP:CHAR;
10 PROCEDURE TERM;
11 PROCEDURE FACTOR;
12 BEGIN
13 IF CH=Ξ(Ξ THEN
14 FIND:
15 EXPRESSION;
16 ELSE
17 WRITE(CH);
18 ENDIF;
19 FIND
20 END;
21 (*FACTOR*)
22 BEGIN FACTOR;
23 WHILE CH=Ξ*Ξ DO
24 FIND;
25 FACTOR;
26 WRITE(Ξ*Ξ)
27 ENDWHILF
28 END;
29 (*TERM*)
30 BEGIN
31 TERM;
32 WHILE (CH=Ξ+Ξ)OR(CH=Ξ-Ξ) DO
33 OP:=CH;
34 FIND;
35 TERM;
36 WRITE(OP):
37 END WHILF
38 END;
39 (*EXPRESSION*)
40 BEGIN
41 FIND;
42 REPEAT
43 WRITE(Ξ Ξ);
44 EXPRESSION;
45 WRITELN
46 UNTIL CH=Ξ.Ξ
47 END.
```

Figure 3.22. Example of Input Text Listing from Verifiable PASCAL Preprocessor



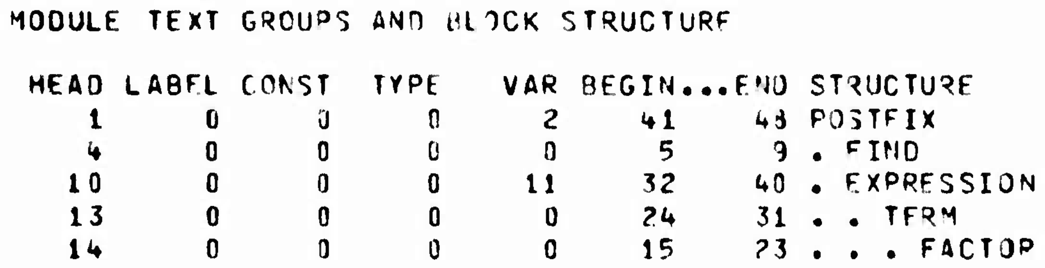
```
000006 PROGFAM POSTFIX ( INPUT , OUTPUT ) ;
000454 VAR
000464 CH : CHAR ;
000465 ASSEPT % BOOLEAN ;
000466 PROCEDURE FIND ;
000003 VAR
000003 ASSERT % BOOLEAN ;
000004 BEGIN
000004 REPEAT
000006 BEGIN
000006 READ ( CH )
000013 END
000013 UNTIL ( CH <> ≡ ≡ ) AND NOT EOLN ( INPUT )
000015 END ;
000025 PROCEDURE EXPRESSION ;
000003 VAR
000003 OP : CHAR ;
000004 ASSEPT % BOOLEAN ;
000005 PROCEDURE TERM ;
000003 VAR
000003 ASSEPT % BOOLEAN ;
000004 PROCFCURE FACTOR ;
000003 VAR
000003 ASSEPT % BOOLEAN ;
000004 BEGIN
000004 IF CH = ≡(≡ THEN
000010 BEGIN
000010 FIND ;
000011 EXPRESSION ;
000012 END
000012 ELSE
000013 BEGIN
000013 WRITE ( CH ) ;
000017 END (*ENDIF *) ;
000017 FIND
000017 END ;
000026 BEGIN
000026 FACTCP ;
000007 WHILF CH = ≡*≡ DO
000011 BEGIN
000011 FIND ;
000012 FACTCP ;
000014 WRITE ( ≡*≡ )
000016 ENC (*ENDWHILE *)
000015 END ;
000025 BEGIN
000025 TERM ;
000007 WHILE ( CH = ≡+≡ ) OR ( CH = ≡-≡ ) DO
000014 BEGIN
000014 OP := CH ;
000017 FIND ;
000020 TERM ;
000022 WRITE ( OP ) ;
000026 END (*END WHILE *)
000026 END ;
000037 BEGIN
000037 FIND ;
000023 REPEAT
000023 BEGIN
000023 WRITE ( ≡ ≡ ) ;
000025 EXPRESSION ;
000026 WRITFLN
000026 FND
000027 UNTIL CH = ≡.≡
000027 END .
```

Figure 3.23. Example of Translated Text Produced by Verifiable PASCAL Preprocessor



```
 1 M    PROGRAM POSTFIX ( INPUT , OUTPUT ) ;                                      .POSTFIX

 2 V    VAR                                                                       .          2
 3 V         CH : CHAR ;                                                          .          3

 4 M    PROCEDURE FIND ;                                                          ..FIND
                                                                  STATEMENT PART OF FIND
 5 S  0 BEGIN                                                                     ..         2
 6 S  0      REPEAT                                                               ..         3
 7 S  1      . READ ( CH )                                                        ..         4
 8 S  0      UNTIL ( CH <> Ξ Ξ ) AND NOT EOLN ( INPUT )                           ..         5
 9 S  0 END ;                                                                     ..         6

10 M    PROCEDURE EXPRESSION ;                                                    ..EXPRESSION

11 V    VAR                                                                       ..         2
12 V         OP : CHAR ;                                                          ..         3

13 M    PROCEDURE TERM ;                                                          ...TERM

14 M    PROCEDURE FACTOR ;                                                        ....FACTOR
                                                                  STATEMENT PART OF FACTOR
15 S  0 BEGIN                                                                     ....       2
16 S  0      IF CH = Ξ(Ξ THEN                                                     ....       3
17 S  1      . FIND ;                                                             ....       4
18 S  1      . EXPRESSION ;                                                       ....       5
19 S  0      ELSE                                                                 ....       6
20 S  1      . WRITE ( CH ) ;                                                     ....       7
21 S  0      ENDIF ;                                                              ....       8
22 S  0      FIND                                                                 ....       9
23 S  0 END ; (*FACTOR*)                                                          ....      10
                                                                  STATEMENT PART OF TERM
24 S  0 BEGIN                                                                     ...        2
25 S  0      FACTOR ;                                                             ...        3
26 S  0      WHILE CH = Ξ*Ξ DO                                                    ...        4
27 S  1      . FIND ;                                                             ...        5
28 S  1      . FACTOR ;                                                           ...        6
29 S  1      . WRITE ( Ξ*Ξ )                                                      ...        7
30 S  0      ENDWHILE                                                             ...        8
31 S  0 END ; (*TERM*)                                                            ...        9
                                                                  STATEMENT PART OF EXPRESSION
32 S  0 BEGIN                                                                     ..         4
33 S  0      TERM ;                                                               ..         5
34 S  0      WHILE ( CH = Ξ+Ξ ) OR ( CH = Ξ-Ξ ) DO                                ..         6
35 S  1      . OP := CH ;                                                         ..         7
36 S  1      . FIND ;                                                             ..         8
37 S  1      . TERM ;                                                             ..         9
38 S  1      . WRITE ( OP ) ;                                                     ..        10
39 S  0      END WHILE                                                            ..        11
40 S  0 END ; (*EXPRESSION*)                                                      ..        12
                                                                  STATEMENT PART OF POSTFIX
41 S  0 BEGIN                                                                     .          4
42 S  0      FIND ;                                                               .          5
43 S  0      REPEAT                                                               .          6
44 S  1      . WRITE ( Ξ Ξ ) ;                                                    .          7
45 S  1      . EXPRESSION ;                                                       .          8
46 S  1      . WRITELN                                                            .          9
47 S  0      UNTIL CH = Ξ.Ξ                                                       .         10
48 S  0 END .                                                                     .         11
```

Figure 3.24. Example of Indented Source Listing from Verifiable PASCAL Preprocessor



```
MODULE  TEXT  GROUPS  AND  BLOCK  STRUCTURE

 HEAD  LABEL  CONST   TYPE    VAR  BEGIN...END  STRUCTURE
    1      0      0      0      2     41     48  POSTFIX
    4      0      0      0      0      5      9  . FIND
   10      0      0      0     11     32     40  . EXPRESSION
   13      0      0      0      0     24     31  . . TERM
   14      0      0      0      0     15     23  . . . FACTOR
```

Figure 3.25. Example of Source Text Directory and Module Structure Report

and module names shown on the indented listing. The directory report (Fig. 3.25) shows the text-wide statement number which begins each text group for each module and the module block structure depicting the scope of the module.

The user may select from various options by preceding the text with an OPTIONS statement. The keyword ASIS is a mnemonic for "as is." The implemented translation options include:

CONTROL = ON (default), translate Verifiable PASCAL control statements (IF...ORIF...ENDIF, etc.) to standard PASCAL

CONTROL = ASIS, pass control statements without translation

ASSERT = ON (default), translate assertion statements to executable code

ASSERT = OFF, translate assertion statements to comments

ASSERT = ASIS, pass assertion statements without translation

UNITS = OFF (default), translate units qualifiers to comments

UNITS = ASIS, pass units qualifiers without translation

RIGHTS = OFF (default), translate data access rights statements
(INPUT, OUTPUT) to comments

RIGHTS = ASIS, pass rights statements without translation

For translation to executable text the user normally selects the default options. For interface to other tools the user selects the option appropriate for the target tool. Since the analyzer design is not yet complete for the interface, user options for the interface have not been specified.

4 STATIC ANALYSIS

Once the syntax analysis of a program is without errors, the second step in developing verifiable software is static analysis, as shown in Fig. 2.1. The Software Quality Laboratory provides a set of consistency checking techniques that detect a wide variety of errors and are applicable to large software systems. They also detect the incorrect use of variables when assertion statements are added to a program. Data access statements and unit qualifiers provide formalized representations of software specifications and programming assumptions at the source language level of detail. Source language conventions and good programming practices also provide general software specifications. Inconsistencies between the software and its specifications are automatically detected and reported by the static analysis tools in the Software Quality Laboratory.

The types of inconsistencies which are detected include:

- Operations on variables with mismatched physical units
- Variables used prior to being assigned a value or set and not used
- Actual use of a variable which differs from asserted use
- Mismatched data types
- Unreachable statements
- Infinite loop constructs
- Inconsistent actual and formal parameters

The Laboratory provides a command language to control the static analysis tools. Commands may be entered as data input cards during batch processing, or interactively through the Anagraph console.[20] Use of the Software Quality Laboratory commands is described in Sec. 4.1. A static analysis tool which performs units checking by validating arithmetic expressions for consistent physical units is included. The physical units consistency analysis algorithm which is implemented in the Laboratory is discussed in Sec. 4.2. Another tool which performs set and use checking by uncovering possible use before set conditions and similar variable use abnormalities is also included. Section 4.3 describes the algorithm usec to perform variable use/set analysis (data flow analysis). The Software Quality Laboratory provides a common data base which integrates the usefulness of static analysis tools. Set/use analysis automatically generates the actual use of program variables. This allows data access assertions to be compared with actual variable usage. The multiple module data base and program structure information required for set/use analysis allow formal and actual parameter checking as well as structural consistency analysis.

4.1 STATIC ANALYSIS COMMANDS AND REPORTS

The static analysis techniques available in the Software Quality Laboratory include:

- Units checking which validates expressions for consistent physical units.
- Set and use checking which uncovers possible use before set conditions and similar program abnormalities.
- Assertion use versus actual use which checks INPUT and OUTPUT statements against actual usage of these variables within the module and validates that unmentioned global variables are not referenced.
- Type and mode checking which identifies possible misuse of constants and variables in expressions, assignments and invocations.
- Graph checking which identifies possible errors in program control structure such as unreachable code.
- Invocation checking which validates actual invocations against formal declarations; checking for consistency in number of parameters and type and intermodule input/output consistency.

The Software Quality Lab analysis tools are controlled using a command language. Static analysis is executed with the command

```
STATIC
```

Seven options may be selected. Five are selected "ON" by default, and two "OFF". To change any of the default settings, it is necessary to insert the appropriate command from the following list before the command STATIC (default values are underlined):

```
STATIC,UNITS=OFF/ON.
STATIC,SET/USE=OFF/ON.
STATIC,ASSERT/ACTUAL=OFF/ON.
STATIC,MODE=OFF/ON.
STATIC,CALL=OFF/ON.
STATIC,GRAPH=OFF/ON.
STATIC,LOOP=OFF/ON.
```

Separate commands are necessary for changing the default for each type of analysis.

The rest of this section describes the analysis performed by each of the commands, illustrates the kinds of errors which can be detected, and explains the static analysis reports.

4.1.1 Physical Units

Description

Requiring that each local variable and each global variable be specified in terms of the physical units it represents (if any) allows comprehensive checking of the consistency of units. This type of checking is particularly relevant to technical software where many physical properties are represented and there are many possibilities of confusion over units. Unics can be checked not only in one module, but across two or more modules if each module contains a description of the units for each physical variable it refers to, in the form of an assertion:

```
UNITS (variable list 1 = units expression 1,
       variable list 2 = units expression 2, ...)
```

An inconsistency in units is indicated if unlike units are added, subtracted, or compared. The physical-units analysis compares the right and left side of assignment statements, the right and left side of relational operations, and actual and formal parameters. For convenience in stating UNITS assertions, all constants are assumed to be unitless, except for zero, which will match any units expression. A variable is declared unitless by stating that its units expression is the constant 1, as in UNITS (PI = 1).

Option Selection

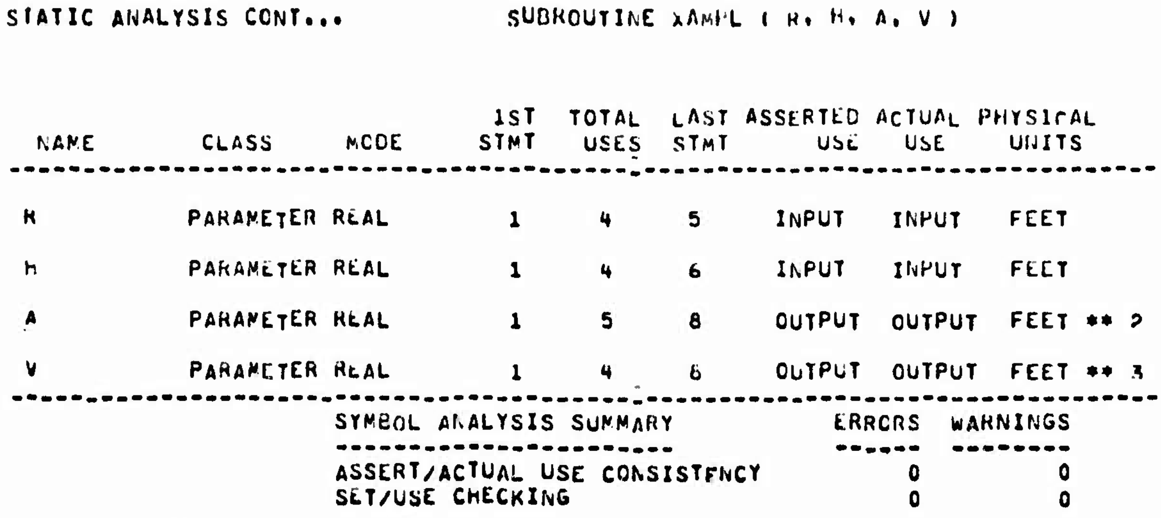
The physical-units analysis is not performed unless the option is selected by the command

```
STATIC,UNITS = ON.
```

Reports

The units asserted to be associated with each variable appear in the Symbol Analysis Summary of the Static Analysis report. This report, for a subroutine called XAMPL, is shown in Fig. 4.1. The last column shows the asserted units for variables R, H, A, and V.

If a physical quantity is asserted to be in units other than actually calculated, the units consistency check will identify such an inconsistency within a given statement, and also indicate interface errors which arise when defined units of parameters passed between routines do not match. All such inconsistencies will be reported in the Statement Analysis Summary of the Static Analysis report.

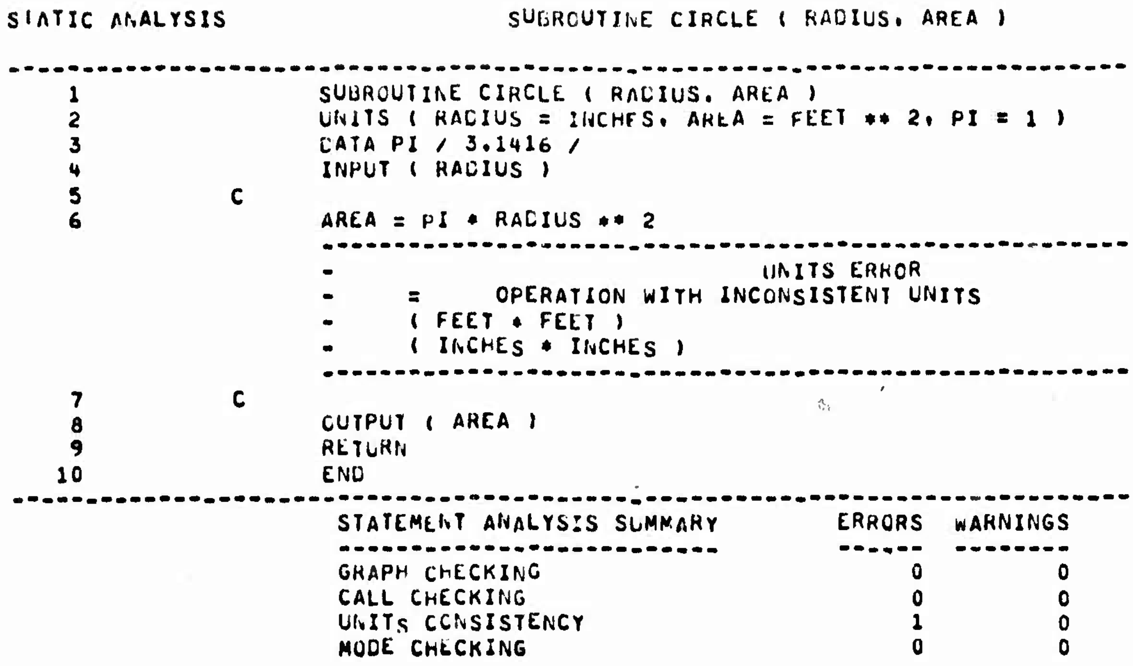


```
STATIC ANALYSIS CONT...              SUBROUTINE XAMPL ( R, H, A, V )


                                          1ST   TOTAL   LAST  ASSERTED ACTUAL PHYSICAL
  NAME        CLASS      MODE           STMT   USES   STMT      USE     USE     UNITS
----------------------------------------------------------------------------------------

  R          PARAMETER REAL               1       4       5      INPUT   INPUT    FEET

  H          PARAMETER REAL               1       4       6      INPUT   INPUT    FEET

  A          PARAMETER REAL               1       5       8      OUTPUT  OUTPUT   FEET ** 2

  V          PARAMETER REAL               1       4       6      OUTPUT  OUTPUT   FEET ** 3
----------------------------------------------------------------------------------------
                      SYMBOL ANALYSIS SUMMARY            ERRORS   WARNINGS
                      -----------------------            ------   --------
                      ASSERT/ACTUAL USE CONSISTENCY         0         0
                      SET/USE CHECKING                      0         0
```

Figure 4.1. Symbol Analysis Summary

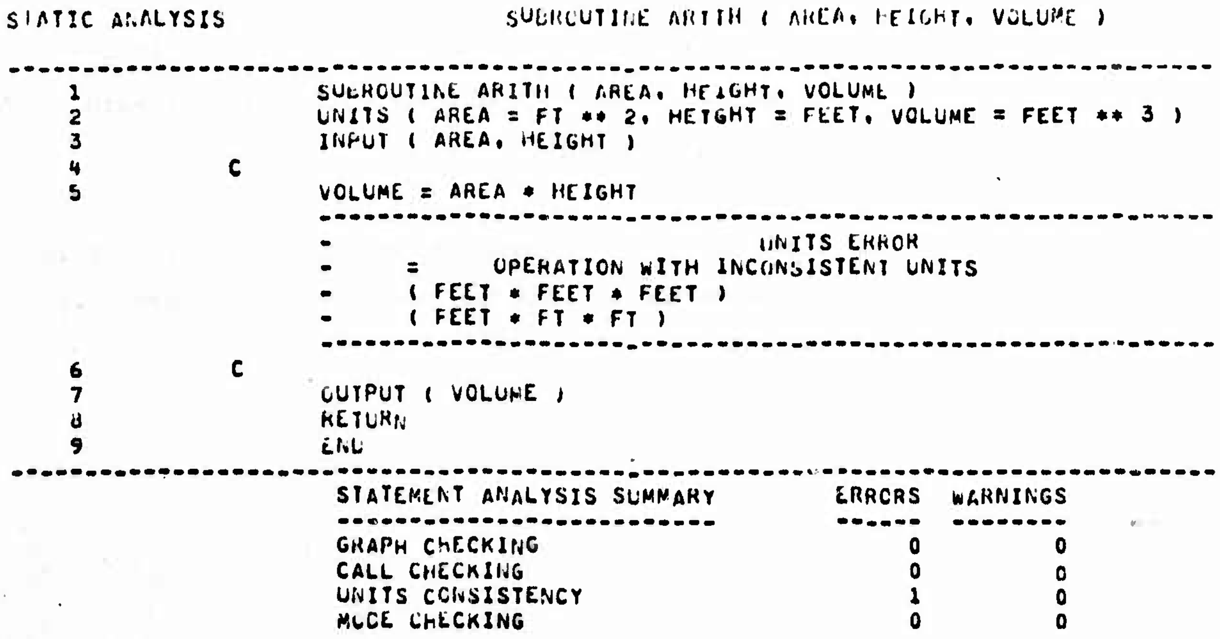
Figure 4.2 shows this report for subroutine CIRCLE. An error message has been printed to show that AREA has been defined as FEET**2 and that the square of RADIUS would be INCHES**2. The units on both sides of an equation must be equivalent.

Figure 4.3 shows another type of units error, in subroutine ARITH. The inconsistency in the spelling of FEET has resulted in the same type of operation error as before.



```
STATIC ANALYSIS                         SUBROUTINE CIRCLE ( RADIUS, AREA )

-------------------------------------------------------------------------------
    1                    SUBROUTINE CIRCLE ( RADIUS, AREA )
    2                    UNITS ( RADIUS = INCHFS, AREA = FEET ** 2, PI = 1 )
    3                    DATA PI / 3.1416 /
    4                    INPUT ( RADIUS )
    5          C
    6                    AREA = PI * RADIUS ** 2
                         --------------------------------------------------------
                         -                               UNITS ERROR
                         -     =     OPERATION WITH INCONSISTENT UNITS
                         -     ( FEET * FEET )
                         -     ( INCHES * INCHES )
                         --------------------------------------------------------
    7          C
    8                    OUTPUT ( AREA )
    9                    RETURN
   10                    END
-------------------------------------------------------------------------------
                          STATEMENT ANALYSIS SUMMARY         ERRORS  WARNINGS
                          --------------------------         ------  --------
                          GRAPH CHECKING                        0        0
                          CALL CHECKING                         0        0
                          UNITS CONSISTENCY                     1        0
                          MODE CHECKING                         0        0
```

Figure 4.2. Units Error Report due to Incorrect Units

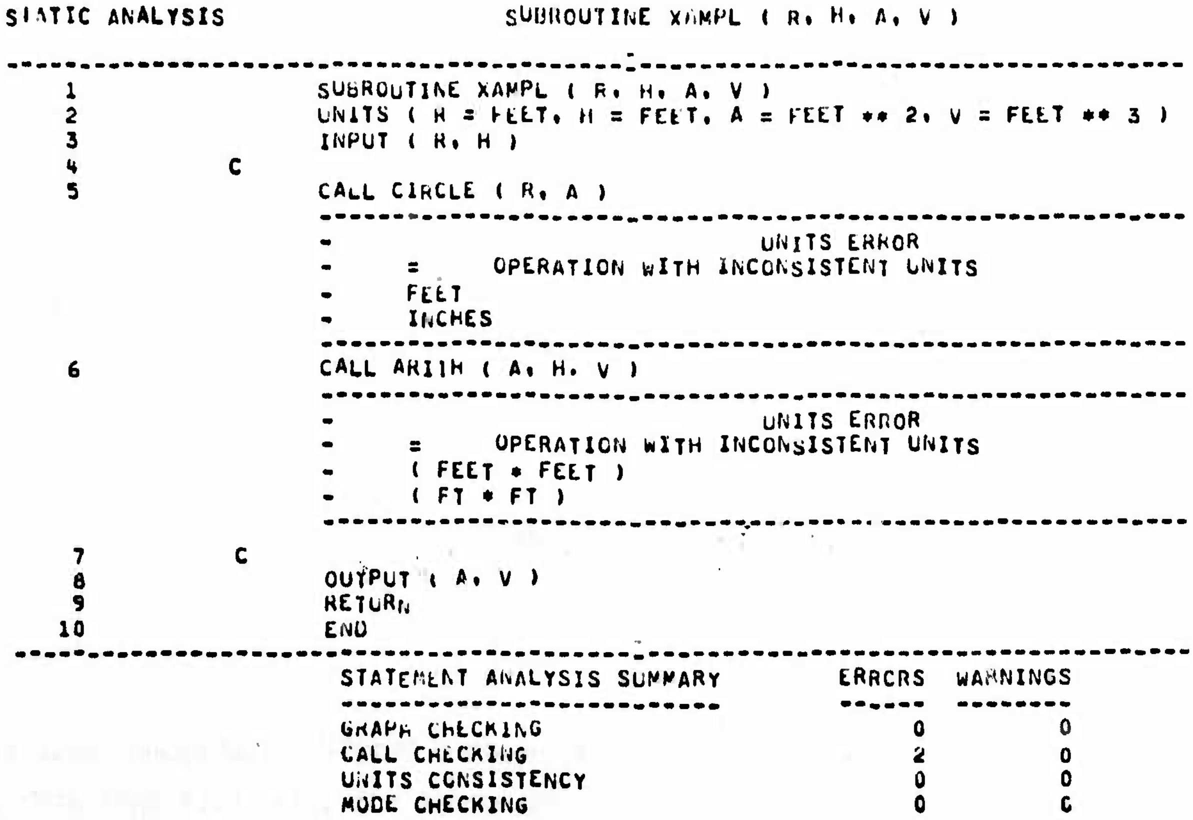


```
STATIC ANALYSIS                         SUBROUTINE ARITH ( AREA, HEIGHT, VOLUME )

-----------------------------------------------------------------------------------
    1                    SUBROUTINE ARITH ( AREA, HEIGHT, VOLUME )
    2                    UNITS ( AREA = FT ** 2, HEIGHT = FEET, VOLUME = FEET ** 3 )
    3                    INPUT ( AREA, HEIGHT )
    4          C
    5                    VOLUME = AREA * HEIGHT
                         ------------------------------------------------------------
                         -                               UNITS ERROR
                         -     =     OPERATION WITH INCONSISTENT UNITS
                         -     ( FEET * FEET * FEET )
                         -     ( FEET * FT * FT )
                         ------------------------------------------------------------
    6          C
    7                    OUTPUT ( VOLUME )
    8                    RETURN
    9                    END
-----------------------------------------------------------------------------------
                          STATEMENT ANALYSIS SUMMARY         ERRORS  WARNINGS
                          --------------------------         ------  --------
                          GRAPH CHECKING                        0        0
                          CALL CHECKING                         0        0
                          UNITS CONSISTENCY                     1        0
                          MODE CHECKING                         0        0
```

Figure 4.3. Units Error Report due to Misspelling

Figure 4.4 shows that the units analysis of the routine XAMPL, which calls CIRCLE and ARITH, has discovered the inconsistency in units of the parameters resulting from the two previous errors:

1. R, correctly specified as FEET, corresponds to RADIUS, which was incorrectly specified as INCHES.
2. A is defined as FEET**2 and AREA as FT**2.



```
STATIC ANALYSIS                     SUBROUTINE XAMPL ( R, H, A, V )

------------------------------------------------------------------------------
    1                    SUBROUTINE XAMPL ( R, H, A, V )
    2                    UNITS ( R = FEET, H = FEET, A = FEET ** 2, V = FEET ** 3 )
    3                    INPUT ( R, H )
    4           C
    5                    CALL CIRCLE ( R, A )
                         -----------------------------------------------------
                         -                                 UNITS ERROR
                         -     =      OPERATION WITH INCONSISTENT UNITS
                         -     FEET
                         -     INCHES
                         -----------------------------------------------------
    6                    CALL ARITH ( A, H, V )
                         -----------------------------------------------------
                         -                                 UNITS ERROR
                         -     =      OPERATION WITH INCONSISTENT UNITS
                         -     ( FEET * FEET )
                         -     ( FT * FT )
                         -----------------------------------------------------
    7           C
    8                    OUTPUT ( A, V )
    9                    RETURN
   10                    END
------------------------------------------------------------------------------
                          STATEMENT ANALYSIS SUMMARY        ERRORS  WARNINGS
                          --------------------------        ------  --------
                          GRAPH CHECKING                        0         0
                          CALL CHECKING                         2         0
                          UNITS CONSISTENCY                     0         0
                          MODE CHECKING                         0         0
```

Figure 4.4. Units Error Report due to Mismatched Parameter/Argument

4.1.2 SET/USE

Description

Just as misuse of physical units is a source of many errors in software systems, so is improper use of program variables. The technique of data flow analysis detects anomalies in the use of variables such as

- Reference to a variable before it has been assigned a value
- Failure to reference a variable after it had been assigned a value

Causes range from simple misspelling to mismatched argument or parameter lists. Not only are these more obvious errors detected, but also more subtle inconsistencies can be found because the flow of data between procedures is examined. When no data flow anomalies have been uncovered, their absence can be assured.

Option Selection

The option to perform the SET/USE analysis of all variables is automatically selected with the command.

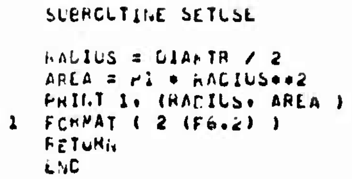
```
STATIC.
```

The option may be turned off by the command

```
STATIC,SET/USE = OFF.
```

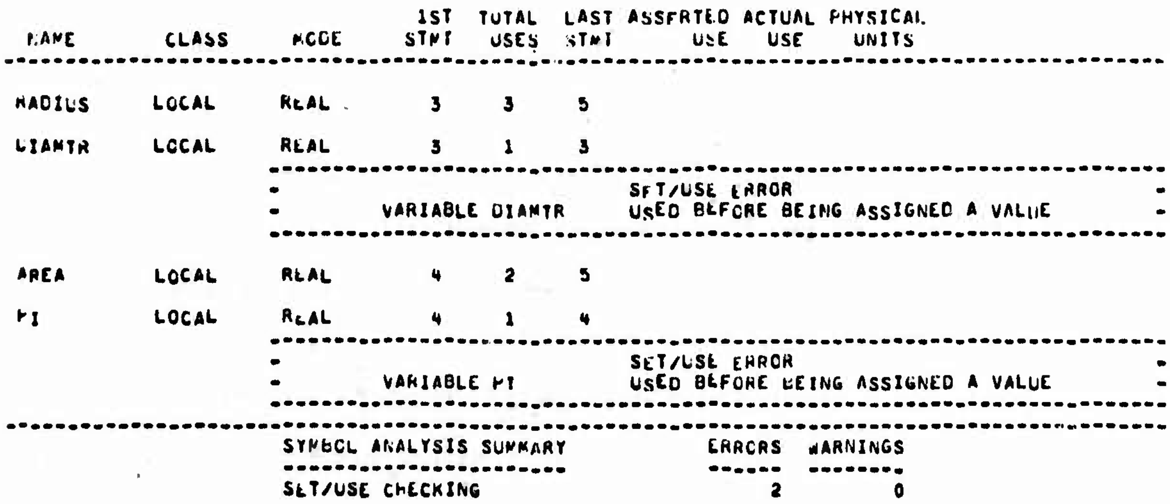
Report

Figure 4.5 is a listing of a subroutine SETUSE. The Symbol Analysis Summary from the Static Analysis report on SET/USE (Fig. 4.6) indicates that neither DIAMTR nor PI had been assigned a value being used.



```
      SUBROUTINE SETUSE

      RADIUS = DIAMTR / 2
      AREA = PI * RADIUS**2
      PRINT 1, (RADIUS, AREA )
    1 FORMAT ( 2 (F6.2) )
      RETURN
      END
```

Figure 4.5. Subroutine SETUSE Statement Listing



```
                                       1ST  TOTAL  LAST ASSERTED ACTUAL PHYSICAL
 NAME         CLASS      MODE         STMT   USES  STMT      USE    USE    UNITS
---------------------------------------------------------------------------------

 RADIUS       LOCAL      REAL            3      3     5

 DIAMTR       LOCAL      REAL            3      1     3
                         --------------------------------------------------------
                         -                              SET/USE ERROR           -
                         -      VARIABLE DIAMTR         USED BEFORE BEING ASSIGNED A VALUE -
                         --------------------------------------------------------

 AREA         LOCAL      REAL            4      2     5

 PI           LOCAL      REAL            4      1     4
                         --------------------------------------------------------
                         -                              SET/USE ERROR           -
                         -      VARIABLE PI             USED BEFORE BEING ASSIGNED A VALUE -
                         --------------------------------------------------------
---------------------------------------------------------------------------------
                          SYMBOL ANALYSIS SUMMARY              ERRORS  WARNINGS
                          -----------------------              ------  --------
                          SET/USE CHECKING                          2         0
```

Figure 4.6. Symbol Analysis Summary with Uninitialized Variables Errors

4.1.3 ASSERT/ACTUAL

Description

A third type of interface consistency checking is to compare the actual use of global variables, as determined by the data flow analysis of variables (see Sec. 4.1.2), with the asserted use.

The asserted use of a variable is stated in INPUT and OUTPUT assertions:

```
INPUT (variable list)
OUTPUT (variable list)
```

An INPUT variable whose value may be changed in the routine should also be included in the OUTPUT variable list.

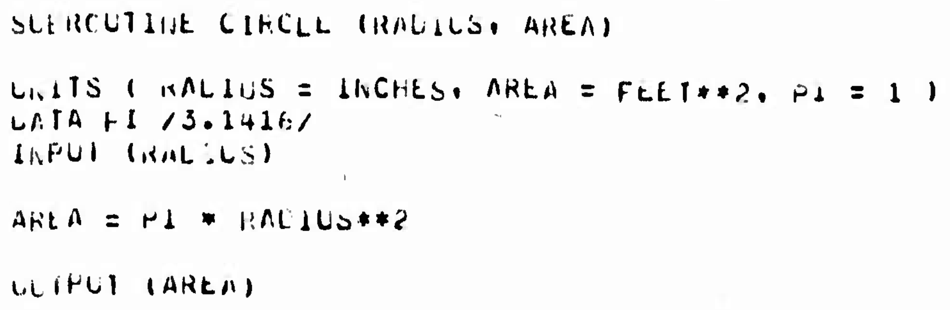
An INPUT assertion states that the variables named:

- Are global variables (either parameters or common variables)
- Will have values whenever the routine is called
- Will not be changed in the routine (unless they have also been listed in the OUTPUT assertion)
- Are the only global variables used in this routine

An OUTPUT assertion states that the variables listed:

- Are global variables (either parameters or common variables)
- Will be assigned a value in the routine
- Will not be used to supply a value to the routine, unless they have also been listed as INPUT variables
- Are the only global variables set in this routine.

A listing of subroutine CIRCLE appears in Fig. 4.7.



```
SUBROUTINE CIRCLE (RADIUS, AREA)

UNITS ( RADIUS = INCHES, AREA = FEET**2, PI = 1 )
DATA PI /3.1416/
INPUT (RADIUS)

AREA = PI * RADIUS**2

OUTPUT (AREA)
```

Figure 4.7. Statement Listing for CIRCLE

In the CIRCLE routine, the assertion

INPUT (RADIUS)

is consistent if the first use of RADIUS on each path in CIRCLE is one of the following:

1. On the right-hand side of an assignment statement
2. Within a decision predicate
3. In a subroutine or function reference where RADIUS is used as input

The assertion

OUTPUT (AREA)

is consistent if any use of AREA in CIRCLE is one of the following:

1. On the left-hand side of an assignment statement
2. In a READ statement
3. In a subroutine or function reference where AREA is used as output.

Assertions about the use of arrays apply to the whole array, not individual elements of the array. This is necessary because data flow analysis does not distinguish the actual use of individual array elements.

Option Selection

The asserted/actual emphasis is not performed unless the option is selected by the command

```
STATIC,ASSERT/ACTUAL=ON.
```

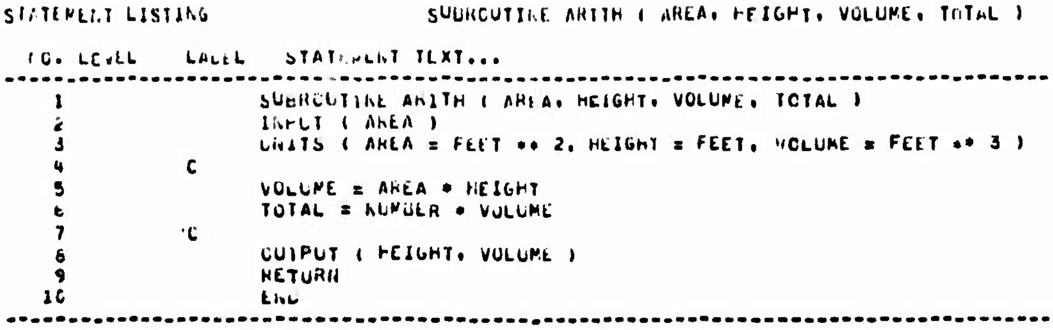
Report

The report provided by this option indicates the number of inconsistencies, either as errors or warnings. For subroutine ARITH, listed in Fig. 4.8, the Symbol Analysis Summary is shown in Fig. 4.9.

The first error occurs because HEIGHT has been listed incorrectly as an OUTPUT variable; its actual use indicates that it is an INPUT parameter which supplies a value for the computation of VOLUME.

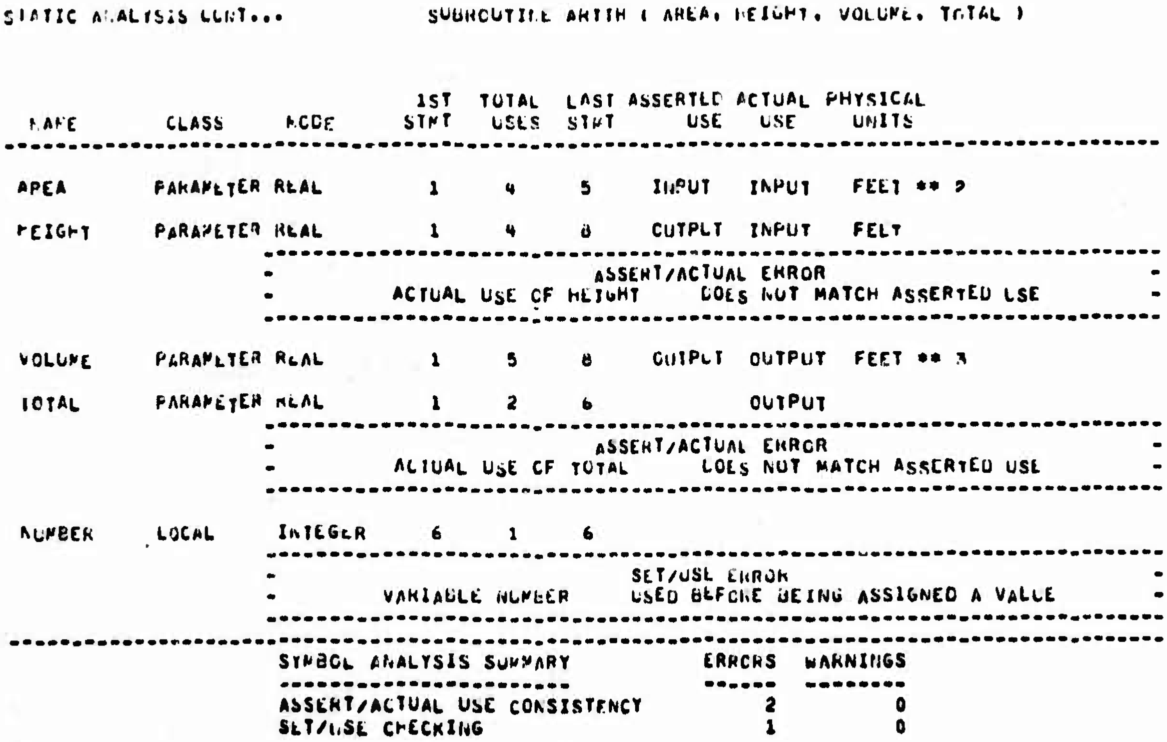
The second error results because TOTAL has not been listed as either INPUT or OUTPUT and hence has no asserted use. Because it is a global variable (a parameter), it must be declared.

The third error is of the SET/USE type described in Sec. 4.1.2. NUMBER is neither in common nor a parameter and so is a local variable; however, it is used in the computation of TOTAL before it has been given a value.



```
STATEMENT LISTING                    SUBROUTINE ARITH ( AREA, HEIGHT, VOLUME, TOTAL )

 NO. LEVEL    LABEL   STATEMENT TEXT...
--------------------------------------------------------------------------------
    1                SUBROUTINE ARITH ( AREA, HEIGHT, VOLUME, TOTAL )
    2                INPUT ( AREA )
    3                UNITS ( AREA = FEET ** 2, HEIGHT = FEET, VOLUME = FEET ** 3 )
    4         C
    5                VOLUME = AREA * HEIGHT
    6                TOTAL = NUMBER * VOLUME
    7         'C
    8                OUTPUT ( HEIGHT, VOLUME )
    9                RETURN
   10                END
--------------------------------------------------------------------------------
```

Figure 4.8. Statement Listing for ARITH



```
STATIC ANALYSIS CONT...          SUBROUTINE ARITH ( AREA, HEIGHT, VOLUME, TOTAL )


                                1ST  TOTAL  LAST ASSERTED ACTUAL PHYSICAL
 NAME         CLASS     MODE    STMT  USES  STMT    USE    USE     UNITS
-----------------------------------------------------------------------------------------

 AREA       PARAMETER REAL        1     4     5    INPUT   INPUT   FEET ** 2

 HEIGHT     PARAMETER REAL        1     4     8    OUTPUT  INPUT   FEET
                      -------------------------------------------------------------------
                      -                          ASSERT/ACTUAL ERROR                     -
                      -         ACTUAL USE OF HEIGHT     DOES NOT MATCH ASSERTED USE     -
                      -------------------------------------------------------------------

 VOLUME     PARAMETER REAL        1     5     8    OUTPUT  OUTPUT  FEET ** 3

 TOTAL      PARAMETER REAL        1     2     6            OUTPUT
                      -------------------------------------------------------------------
                      -                          ASSERT/ACTUAL ERROR                     -
                      -         ACTUAL USE OF TOTAL      DOES NOT MATCH ASSERTED USE     -
                      -------------------------------------------------------------------

 NUMBER      LOCAL    INTEGER     6     1     6
                      -------------------------------------------------------------------
                      -                              SET/USE ERROR                       -
                      -        VARIABLE NUMBER     USED BEFORE BEING ASSIGNED A VALUE    -
                      -------------------------------------------------------------------
-----------------------------------------------------------------------------------------
                       SYMBOL ANALYSIS SUMMARY                ERRORS  WARNINGS
                       -----------------------                ------  --------
                       ASSERT/ACTUAL USE CONSISTENCY               2         0
                       SET/USE CHECKING                            1         0
```

Figure 4.9. Symbol Analysis Summary with Variable Use Assertion Errors

4.1.4 MODE

Description

Potential errors which result from inconsistencies in the mode of variables (REAL, INTEGER, etc.) can be found in the static analysis checking of expressions.

Option Selection

Mode checking is one of the options which is included in the analysis performed by the command, STATIC. To turn off the option

```
        STATIC,MODE=OFF.
```

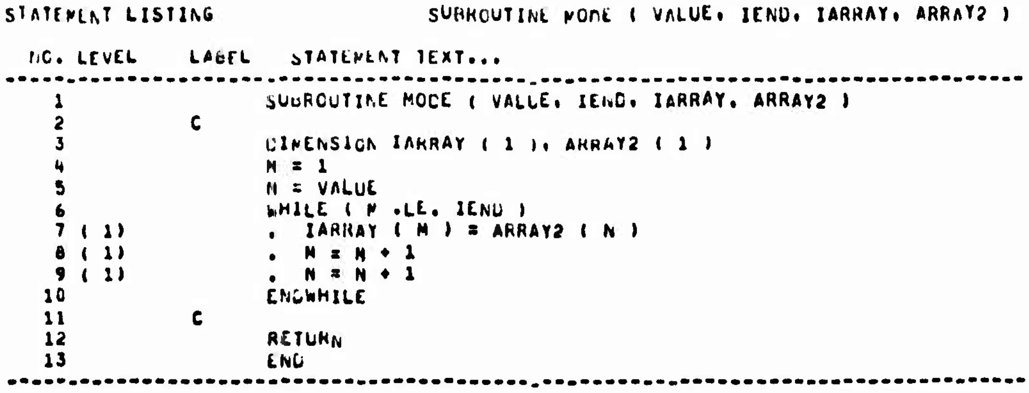
must be commanded before the STATIC command.

Report

The Symbol Analysis Summary of the Static Analysis report lists the name and mode of each variable which has been set or used in a routine.

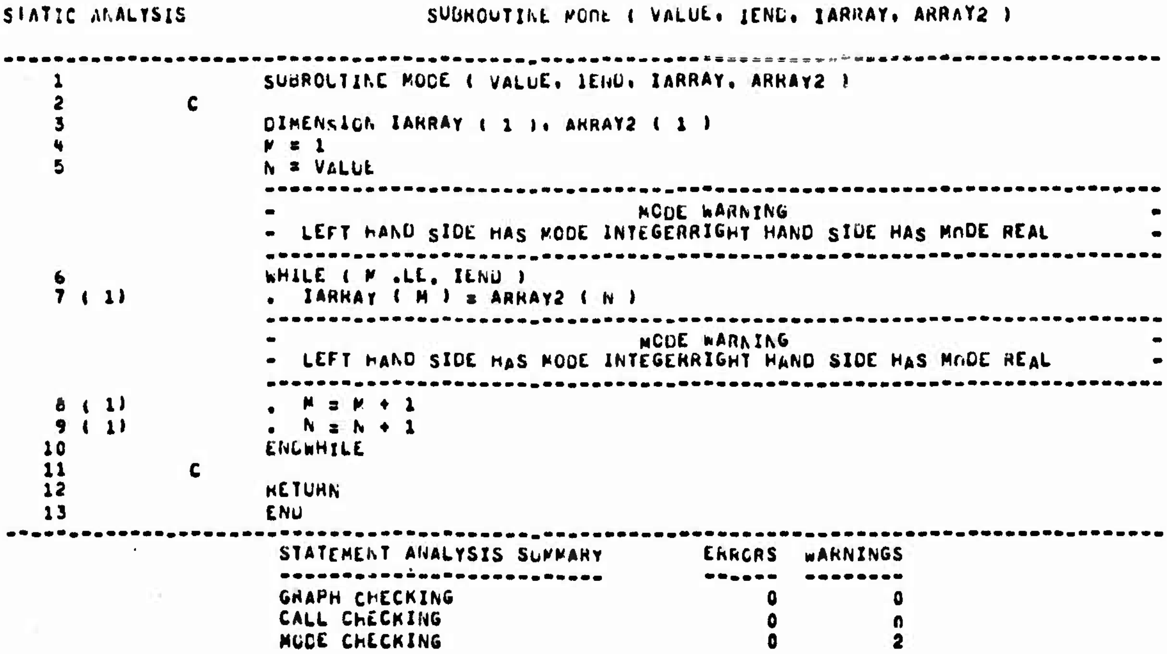
A mode inconsistency causes a warning to be printed immediately following the statement in which it occurs in the listing, and the number of warnings is tabulated in the Statement Analysis Summary.

Figure 4.10 is a listing of subroutine MODE; it contains two mode errors, as can be seen in Fig. 4.11.



```
STATEMENT LISTING                   SUBROUTINE MODE ( VALUE, IEND, IARRAY, ARRAY2 )

 NO. LEVEL     LABEL   STATEMENT TEXT...
-------------------------------------------------------------------------------------
   1                   SUBROUTINE MODE ( VALUE, IEND, IARRAY, ARRAY2 )
   2             C
   3                   DIMENSION IARRAY ( 1 ), ARRAY2 ( 1 )
   4                   M = 1
   5                   N = VALUE
   6                   WHILE ( M .LE. IEND )
   7 ( 1)              .  IARRAY ( M ) = ARRAY2 ( N )
   8 ( 1)              .  M = M + 1
   9 ( 1)              .  N = N + 1
  10                   ENDWHILE
  11             C
  12                   RETURN
  13                   END
-------------------------------------------------------------------------------------
```

Figure 4.10. Statement Listing for Subroutine MODE



```
STATIC ANALYSIS                     SUBROUTINE MODE ( VALUE, IEND, IARRAY, ARRAY2 )

------------------------------------------------------------------------------------------
   1             SUBROUTINE MODE ( VALUE, IEND, IARRAY, ARRAY2 )
   2         C
   3             DIMENSION IARRAY ( 1 ), ARRAY2 ( 1 )
   4             M = 1
   5             N = VALUE
                 --------------------------------------------------------------------------
                 -                          MODE WARNING                                  -
                 -  LEFT HAND SIDE HAS MODE INTEGERRIGHT HAND SIDE HAS MODE REAL          -
                 --------------------------------------------------------------------------
   6             WHILE ( M .LE. IEND )
   7 ( 1)        .  IARRAY ( M ) = ARRAY2 ( N )
                 --------------------------------------------------------------------------
                 -                          MODE WARNING                                  -
                 -  LEFT HAND SIDE HAS MODE INTEGERRIGHT HAND SIDE HAS MODE REAL          -
                 --------------------------------------------------------------------------
   8 ( 1)        .  M = M + 1
   9 ( 1)        .  N = N + 1
  10             ENDWHILE
  11         C
  12             RETURN
  13             END
------------------------------------------------------------------------------------------
                  STATEMENT ANALYSIS SUMMARY        ERRORS  WARNINGS
                  --------------------------        ------  --------
                  GRAPH CHECKING                         0         0
                  CALL CHECKING                          0         0
                  MODE CHECKING                          0         2
```

Figure 4.11. Statement Analysis Summary with Mode Warnings

4.1.5 CALL

Description

Another of the static analysis options specifies interprocedural checking of subroutine or procedure invocations to reveal situations which may lead to errors, such as:

- The number of parameters listed does not agree with those of the routine called.
- The mode of an actual parameter does not match that of the corresponding formal parameter.
- A parameter is listed in the calling argument list as a single, non-subscripted variable but is used in the routine as an array.
- The routine called does not exist in the set of modules being tested.

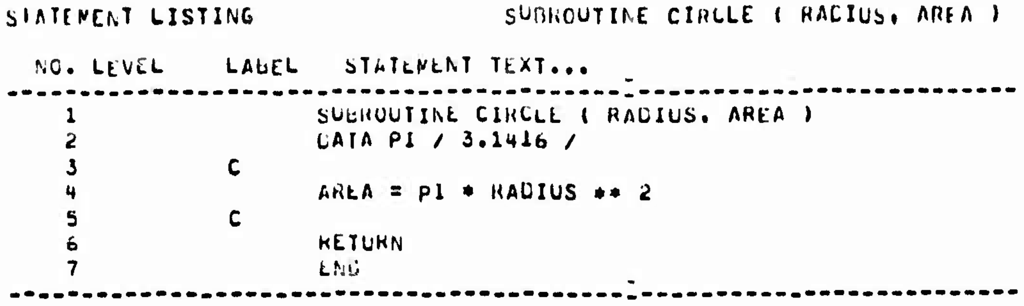
Option Selection

Call checking of parameter lists is automatically included in the analysis specified by the STATIC command. If the option is not wanted, it is turned off by the command

```
STATIC,CALL=OFF.
```

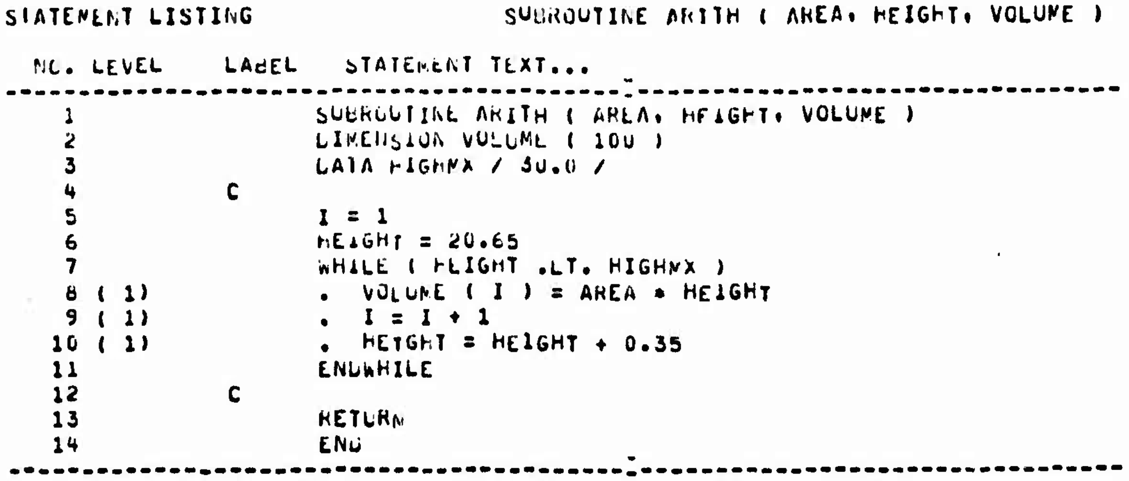
Report

Figures 4.12 and 4.13 are listings of Subroutines CIRCLE and ARITH (somewhat different from those given in Figs. 4.7 and 4.8).



```
STATEMENT LISTING                     SUBROUTINE CIRCLE ( RADIUS, AREA )

  NO. LEVEL     LABEL   STATEMENT TEXT...
------------------------------------------------------------------------
     1                 SUBROUTINE CIRCLE ( RADIUS, AREA )
     2                 DATA PI / 3.1416 /
     3           C
     4                 AREA = PI * RADIUS ** 2
     5           C
     6                 RETURN
     7                 END
------------------------------------------------------------------------
```

Figure 4.12. Statement Listing: Subroutine CIRCLE

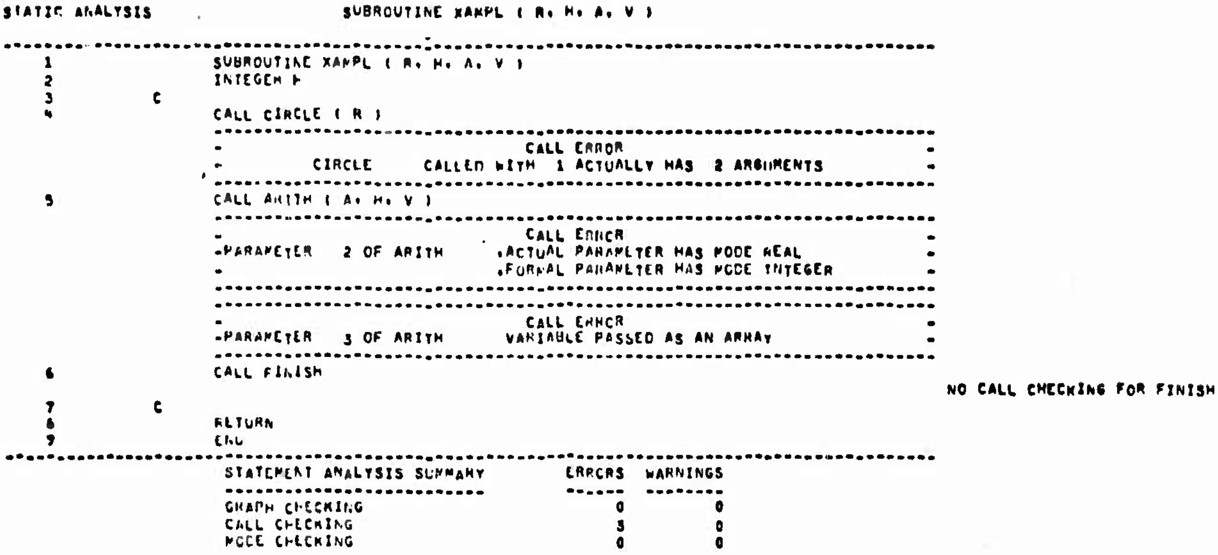


```
STATEMENT LISTING                     SUBROUTINE ARITH ( AREA, HEIGHT, VOLUME )

  NO. LEVEL     LABEL   STATEMENT TEXT...
-------------------------------------------------------------------------------
     1                 SUBROUTINE ARITH ( AREA, HEIGHT, VOLUME )
     2                 DIMENSION VOLUME ( 100 )
     3                 DATA HIGHMX / 30.0 /
     4           C
     5                 I = 1
     6                 HEIGHT = 20.65
     7                 WHILE ( HEIGHT .LT. HIGHMX )
     8 ( 1)            .  VOLUME ( I ) = AREA * HEIGHT
     9 ( 1)            .  I = I + 1
    10 ( 1)            .  HEIGHT = HEIGHT + 0.35
    11                 ENDWHILE
    12           C
    13                 RETURN
    14                 END
-------------------------------------------------------------------------------
```

Figure 4.13. Statement Listing: Subroutine ARITH

A report showing interface inconsistencies among three modules is generated in the Statement Analysis Summary for Subroutine XAMPL, Fig. 4.14.

The first error occurs because CIRCLE has two arguments (RADIUS and AREA) and the invocation has one. Two errors result from the invocation of Subroutine ARITH. The variable H has been declared as an integer, but HEIGHT, the variable in ARITH which corresponds to H , is real. The variable V is a single, non-subscripted variable, but, in ARITH, VOLUME has been dimensioned as an array.



```
STATIC ANALYSIS                    SUBROUTINE XAMPL ( R, H, A, V )

-------------------------------------------------------------------------------------------
    1                 SUBROUTINE XAMPL ( R, H, A, V )
    2                 INTEGER H
    3        C
    4                 CALL CIRCLE ( R )
                      ---------------------------------------------------------------------
                      -                            CALL ERROR                             -
                      -         CIRCLE     CALLED WITH  1 ACTUALLY HAS  2 ARGUMENTS       -
                      ---------------------------------------------------------------------
    5                 CALL ARITH ( A, H, V )
                      ---------------------------------------------------------------------
                      -                            CALL ERROR                             -
                      -PARAMETER   2 OF ARITH    ,ACTUAL PARAMETER HAS MODE REAL          -
                      -                          ,FORMAL PARAMETER HAS MODE INTEGER       -
                      ---------------------------------------------------------------------
                      ---------------------------------------------------------------------
                      -                            CALL ERROR                             -
                      -PARAMETER   3 OF ARITH     VARIABLE PASSED AS AN ARRAY             -
                      ---------------------------------------------------------------------
    6                 CALL FINISH
                                                                                             NO CALL CHECKING FOR FINISH
    7        C
    8                 RETURN
    9                 END
-------------------------------------------------------------------------------------------
                       STATEMENT ANALYSIS SUMMARY        ERRORS  WARNINGS
                       --------------------------        ------  --------
                       GRAPH CHECKING                         0         0
                       CALL CHECKING                          3         0
                       MODE CHECKING                          0         0
```

Figure 4.14. Statement Analysis Summary Showing Calling Errors

Subroutine FINISH was not included in the set of routines examined in this static analysis. Although this is not an actual error, a message is printed on the right side of the report as a reminder.

4.1.6 Unreachable Statements

Description

An obvious consistency check is that of structural consistency. The program graph for each module can be checked to see that all statements are reachable from each statement. Unreachable statements represent extra overhead in terms of memory space required for a module, while statements from which the exit cannot be reached represent potentially catastrophic system failures.

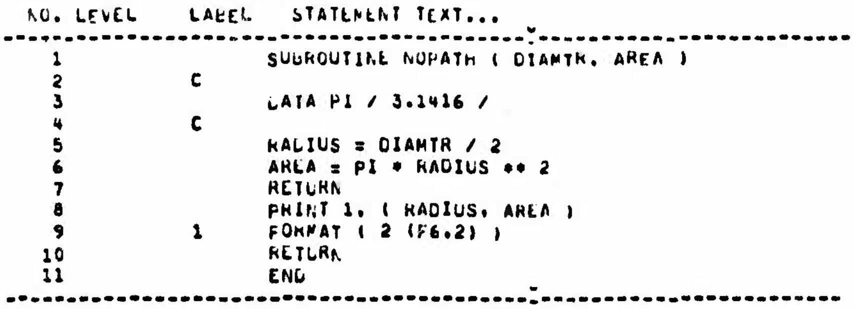
Option Selection

The checking for unreachable statements is automatically included in the analysis specified by the STATIC command. If the option is not wanted, it is turned off by the command

```
STATIC,GRAPH=OFF
```

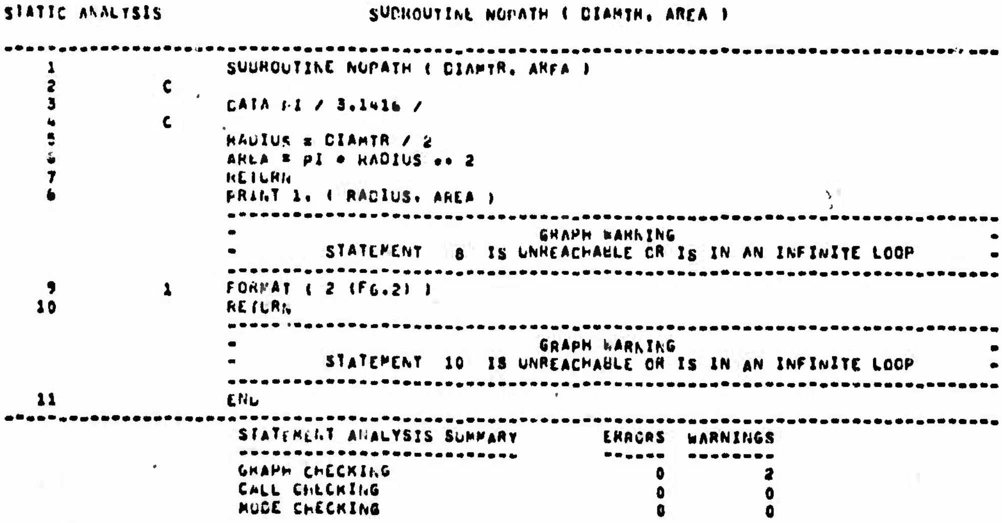
Report

Figure 4.15 is a statement listing of NOPATH. In this subroutine, Statement 7 is a RETURN statement. The two executable statements which follow it are unreachable, and a warning message is printed for each in the Statement Analysis Summary (Fig. 4.16).



```
 NO. LEVEL     LABEL   STATEMENT TEXT...
----------------------------------------------------------------------------
   1                      SUBROUTINE NOPATH ( DIAMTR, AREA )
   2           C
   3                      DATA PI / 3.1416 /
   4           C
   5                      RADIUS = DIAMTR / 2
   6                      AREA = PI * RADIUS ** 2
   7                      RETURN
   8                      PRINT 1, ( RADIUS, AREA )
   9           1          FORMAT ( 2 (F6.2) )
  10                      RETURN
  11                      END
----------------------------------------------------------------------------
```

Figure 4.15. Statement Listing of Subroutine NOPATH



```
STATIC ANALYSIS                    SUBROUTINE NOPATH ( DIAMTR, AREA )

---------------------------------------------------------------------------------------
   1           C         SUBROUTINE NOPATH ( DIAMTR, AREA )
   2           C
   3                     DATA PI / 3.1416 /
   4           C
   5                     RADIUS = DIAMTR / 2
   6                     AREA = PI * RADIUS ** 2
   7                     RETURN
   8                     PRINT 1, ( RADIUS, AREA )
                         --------------------------------------------------------------
                         -                        GRAPH WARNING                        -
                         -        STATEMENT   8  IS UNREACHABLE OR IS IN AN INFINITE LOOP     -
                         --------------------------------------------------------------
   9           1         FORMAT ( 2 (F6.2) )
  10                     RETURN
                         --------------------------------------------------------------
                         -                        GRAPH WARNING                        -
                         -        STATEMENT  10  IS UNREACHABLE OR IS IN AN INFINITE LOOP     -
                         --------------------------------------------------------------
  11                     END
---------------------------------------------------------------------------------------
                          STATEMENT ANALYSIS SUMMARY        ERRORS  WARNINGS
                          --------------------------        ------  --------
                          GRAPH CHECKING                       0        2
                          CALL CHECKING                        0        0
                          MODE CHECKING                        0        0
```

Figure 4.16. Statement Analysis Summary with Unreachable Statement Errors

4.1.7 Loop Constructs

Description

One of the most frustrating and common errors is an infinite loop construct. A check on structural consistencies determines if this possibility exists.

Option Selection

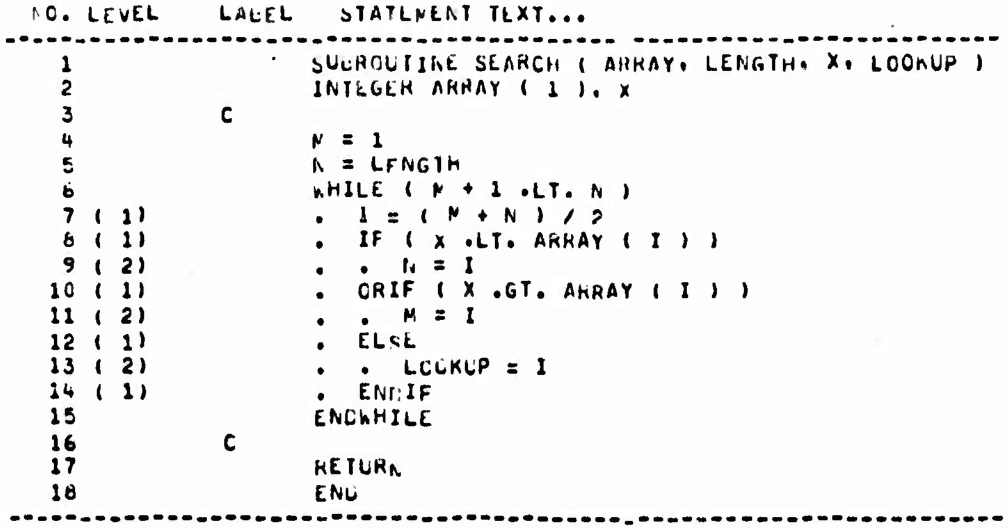
This option is on when the STATIC analysis is performed unless it has been turned off with the command

```
STATIC,LOOP=OFF.
```

Report

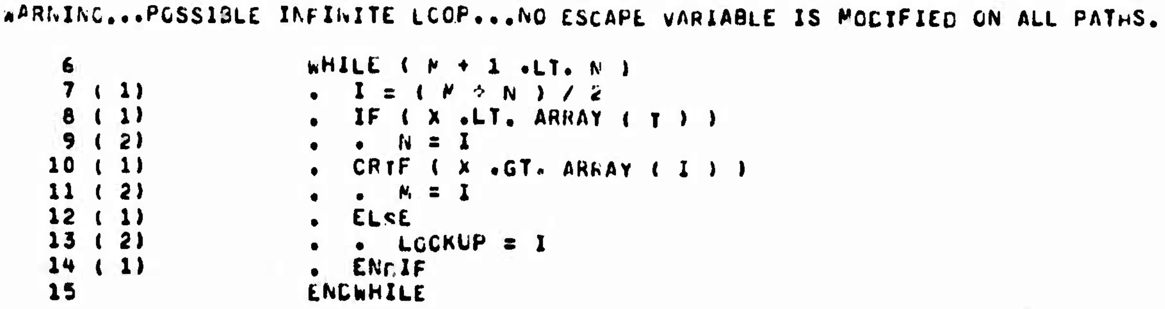
No report is generated unless the possibility of an infinite loop construct exists. If such a construct has been located, a loop analysis report is generated containing a warning message and the statements of the loop in question.

Some errors of this type are not immediately obvious and, therefore, are difficult to detect. One such error is in subroutine SEARCH, listed in Fig. 4.17. Figure 4.18 shows the report, which includes the portion of the code where there may be an infinite loop. The infinite loop would occur when the ELSE path is taken: LOOKUP is set to I , but neither M nor N is modified, so that the conditions of the loop would be infinitely repeated.



```
NO. LEVEL    LABEL   STATEMENT TEXT...
----------------------------------------------------------------------
  1                  SUBROUTINE SEARCH ( ARRAY, LENGTH, X, LOOKUP )
  2                  INTEGER ARRAY ( 1 ), X
  3            C
  4                  M = 1
  5                  N = LENGTH
  6                  WHILE ( M + 1 .LT. N )
  7 ( 1)             .  I = ( M + N ) / 2
  8 ( 1)             .  IF ( X .LT. ARRAY ( I ) )
  9 ( 2)             .  .  N = I
 10 ( 1)             .  ORIF ( X .GT. ARRAY ( I ) )
 11 ( 2)             .  .  M = I
 12 ( 1)             .  ELSE
 13 ( 2)             .  .  LOOKUP = I
 14 ( 1)             .  ENDIF
 15                  ENDWHILE
 16            C
 17                  RETURN
 18                  END
----------------------------------------------------------------------
```

Figure 4.17. Statement Listing of Subroutine SEARCH



```
WARNING...POSSIBLE INFINITE LOOP...NO ESCAPE VARIABLE IS MODIFIED ON ALL PATHS.

   6                 WHILE ( M + 1 .LT. N )
   7 ( 1)            .  I = ( M + N ) / 2
   8 ( 1)            .  IF ( X .LT. ARRAY ( I ) )
   9 ( 2)            .  .  N = I
  10 ( 1)            .  ORIF ( X .GT. ARRAY ( I ) )
  11 ( 2)            .  .  M = I
  12 ( 1)            .  ELSE
  13 ( 2)            .  .  LOOKUP = I
  14 ( 1)            .  ENDIF
  15                 ENDWHILE
```

Figure 4.18. Loop Analysis Warning Report

4.2 PHYSICAL UNITS CONSISTENCY ANALYSIS

Physical units checking is an excellent example of consistency analysis using a partial program specification at the source language level of detail. The units tester uses UNITS statements (described in Sec. 3) to associate specified physical units with program variables. If program variables RGB, RHO, GR2, and BETA are specified by

```
UNITS(RGB = 1/FT)
UNITS(RHO = (SEC**2)/LBS)
UNITS(GR2 = FT/(SEC**2))
UNITS(BETA = LBS/(FT**2))
```

then the computation of RGB as

```
RGB = (-RHO*GR2)/BETA
```

is consistent. This can be verified by substitution of unit qualifiers and simplification. The right-hand side of this assignment statement becomes

```
(-((SEC**2)/LBS)*FT/(SEC**2))/(LBS/(FT**2))
```

cancelling the (SEC**2) terms results in

```
(-FT/LBS)/(LBS/FT**2)
```

cancelling common FT and LBS terms, and dropping the minus sign yields

```
(1/FT)
```

which matches the units description of the left-hand side of the assignment statement.

An algorithm to automatically perform this substitution and simplification process has been implemented in the Software Quality Lab. The physical units checking process transforms each arithmetic expression into a tree with unit qualifiers as nodes. Unit qualifiers associated with variables are specified with UNITS statements. Figure 4.19 indicates physical units assertions, the statement to be analyzed and the units tree which results.

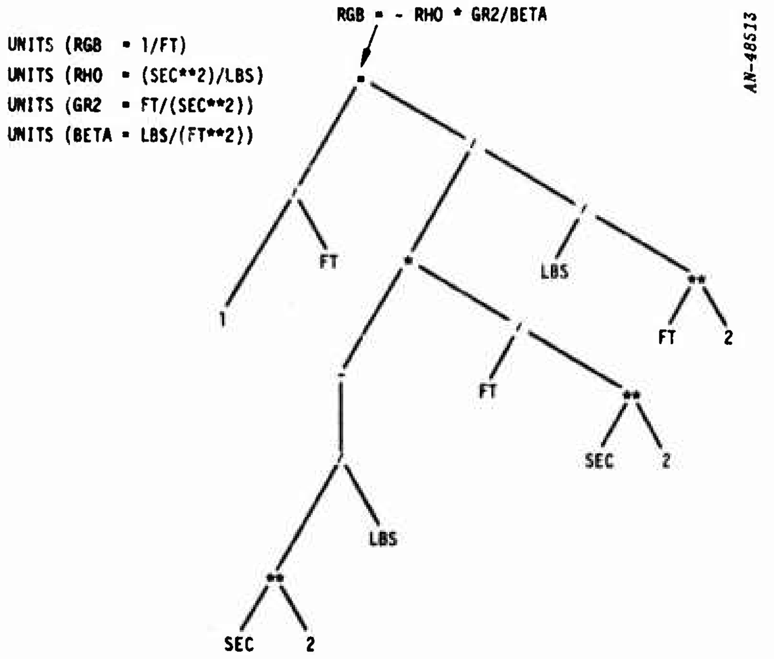


Figure 4.19. Physical Units Tree

The physical units simplification and normalization rules which are applied by the algorithm are shown in Fig. 20 and Fig. 21. Since like units can be added or subtracted, the addition rule in Fig. 20 removes addition and subtraction operators from the physical units tree. Like units can also be compared. The relational rule in Fig. 4.20 replaces relational operators (greater than, equal, not equal, etc.) with the value TRUE. The addition, subtraction, or comparison of unlike physical units immediately results in a units inconsistency. Additional simplification rules include cancellation of like units in a numerator and denominator, dropping unary minus and unary plus operators, and logical operation simplification.

Normalization rules are limited to multiplication, division, and exponentation operators since all other operators are simplified out. The cummulative property of multiplication is used to lexically order

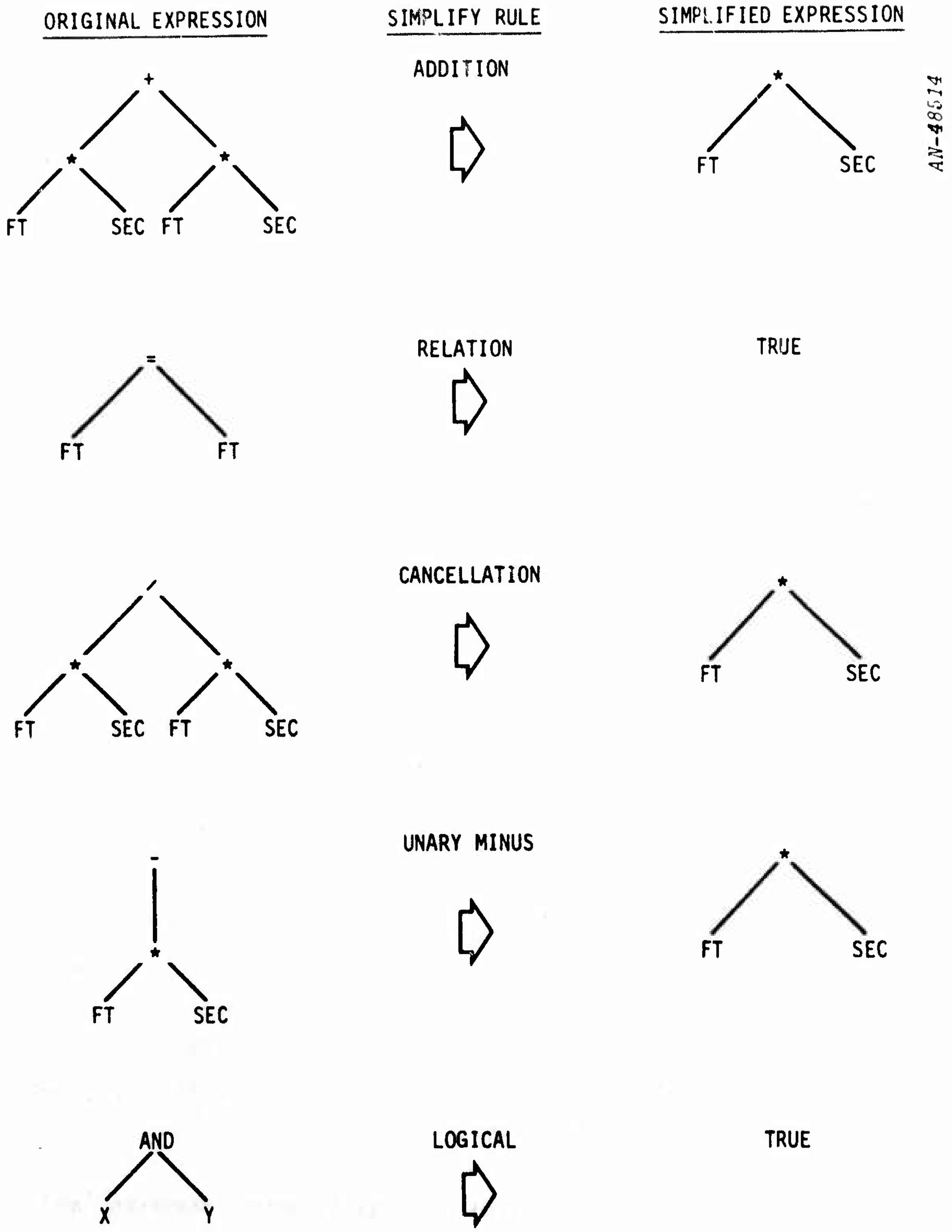


Figure 4.20. Physical Units Simplification Rules

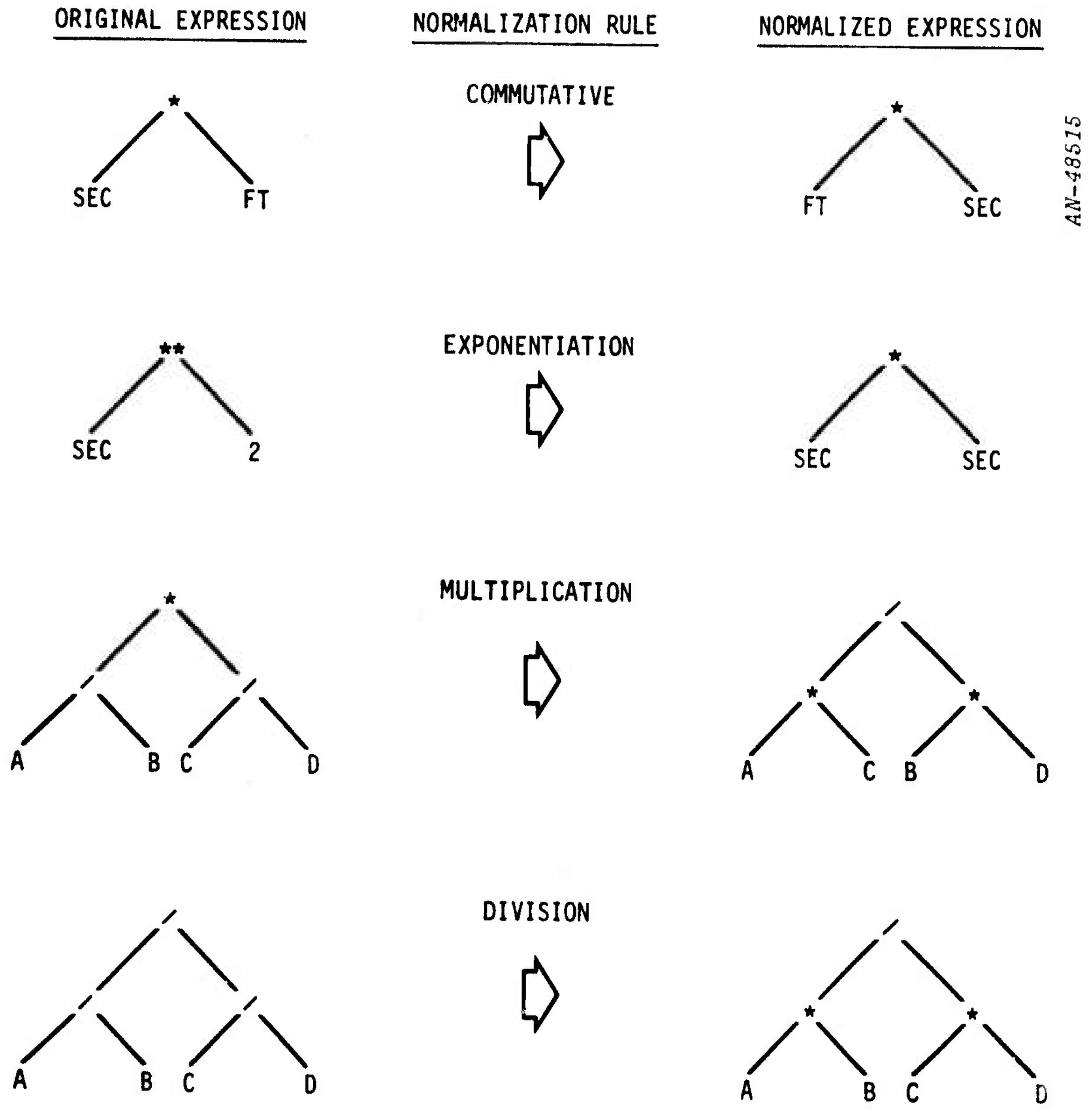


Figure 4.21. Physical Units Normalization Rules

all product terms (Fig. 4.21). Exponentiation operations are converte, to an equivalent product term (this limits the analysis to integer exponents). The normal form for units expressions is a quotient of ordered product terms. The multiplication and division normalizations in Fig. 4.21 maintain this form by moving all division operators to the root of the physical units tree being analyzed.

The physical units consistency analyzer applies these rules as the physical units tree is walked. Each subtree of a nonterminal node must be in normalized, simplified form before the rules are applied to that nonterminal node. A recursive description of the algorithm is given in Fig. 4.22.

```
PROCEDURE UNITS SIMPLIFICATION

IF THE CURRENT NODE IS A NONTERMINAL
.   PERFORM UNITS SIMPLIFICATION ON LEFT SUB-TREE
.   IF THE CURRENT NODE IS NOT A UNARY OPERATOR
.   .   PERFORM UNITS SIMPLIFICATION ON RIGHT SUB-TREE
.   END IF
.   IF THE CURRENT NODE IS A +,-,OR RELATIONAL OPERATOR
.   .   IF THE LEFT SUB-TREE EQUALS THE RIGHT SUB-TREE
.   .   .   IF THE CURRENT NODE IS A +,OR - OPERATOR
.   .   .   .   PERFORM ADDITION SIMPLIFICATION
.   .   .   ELSE
.   .   .   .   PERFORM RELATIONAL SIMPLIFICATION
.   .   .   END IF
.   .   ELSE
.   .   .   A UNITS INCONSISTENCY HAS BEEN FOUND.......RETURN
.   .   END IF
.   ORIF THE CURRENT NODE IS A * OR / OPERATOR
.   .   IF EITHER SUBTREE IS A / OPERATOR
.   .   .   PERFORM MULTIPLICATION OR DIVISION NORMALIZATION
.   .   .   PERFORM COMMUTATIVE NORMALIZATION OF NUMERATOR
.   .   .   PERFORM COMMUTATIVE NORMALIZATION OF DENOMINATOR
.   .   .   PERFORM CANCELATION SIMPLIFICATION
.   .   ORIF CURRENT NODE IS A / OPERATOR
.   .   .   PERFORM COMMUTATIVE NORMALIZATION OF NUMERATOR
.   .   .   PERFORM COMMUTATIVE NORMALIZATION OF DENOMINATOR
.   .   .   PERFORM CANCELLATION SIMPLIFICATION
.   .   ELSE
.   .   .   PERFORM COMMUTATIVE NORMALIZATION
.   .   END IF
.   ORIF CURRENT NODE IS A UNARY PLUS OR MINUS
.   .   PERFORM UNARY SIMPLIFICATION
.   ORIF CURRENT NODE IS A LOGICAL OPERATOR
.   .   PERFORM LOGICAL SIMPLIFICATION
.   ORIF CURRENT NODE IS AN ** OPERATOR
.   .   IF EXPONENT IS AN INTEGER CONSTANT
.   .   .   PERFORM EXPONENT NORMALIZATION
.   .   ELSE
.   .   .   UNABLE TO COMPLETE SIMPLIFICATION..........RETURN
.   .   END IF
.   END IF
END IF
RETURN
END
```

Figure 4.22. Physical Units Consistency Analysis Algorithm

4.3 DATA FLOW ANALYSIS

Improper use of program variables is a major souce of errors in large software systems. The technique of data flow analysis[21] detects anomalies in the use of variables such as references to a variable before it has been assigned a value.

The results of data flow analysis technique in the Software Quality Laboratory are similar to that performed by the DAVE system. A leaf-first analysis of the system calling tree is performed. While the DAVE system analyzes each module with a variable-by-variable depth first search, the Software Quality Laboratory performs a parallel analysis of all variables in a module. The resulting computation time can be several orders of magnitude faster with the parallel analysis technique.

Both techniques are applicable to FORTRAN source code. DAVE is limited to ANSII standard FORTRAN. The Software Quality Laboratory can analyze common extensions to ANSII FORTRAN, including a dialect of structured FORTRAN(IFTRAN). The Software Quality Laboratory techniques include optional consistency checking of actual variables usage and desired variable usage. DAVE does not provide this additional check.

Data flow analysis classifies the usage of all program variables. A variable is used as output (to receive a value) if any of its uses is one of the following:

1. On the left-hand side of an assignment statement
2. In an initialization statement such as a DATA statement
3. In a READ statement

A variable is used as input (to supply a value) if its first use on some path is one of the following:

1. On the right-hand side of an assignment statement
2. Within a decision predicate
3. Within a WRITE statement.

A path originates at the entry to the software system, follows one of many structurally possible routes thru the software, and terminates at an exit from the software system. A path will, in general, go through more than one module of a multi-module software system.

A naive approach to data flow analysis is to identify all possible paths and compute the use of each program variable on each path. This approach quickly runs into the combinatorics problem of too many paths in even moderately sized software systems. Several techniques are available to dramatically improve the efficiency of data flow analysis. The first technique uses a software system's intermodule calling structure. The calling structure is represented as a tree whose root is the main module in the software system. The calling tree is then analyzed in a bottom-up fashion. Modules which do not invoke any other modules (leaf modules) are analyzed first. Then modules which invoke only leaf modules are analyzed, etc. After a module has been analyzed, it can be represented in terms of its variables and how they are used rather than as a set of statements with an inherent structure. Each module is only analyzed once when this techique is used. A similar technique allows each statement in each module to be analyzed only once during a complete data flow analysis for all program variables. This technique is based on (but not limited to) a well-structured, single-entry, single-exit program graph. Data flow analysis proceeds sequentially thru each statement in a well structured program. The use state of each program variable is updated as statements are sequentially analyzed. Control statements from which several parallel paths originate cause parallel use states to be computed. Control statements at which several parallel paths rejoin cause parallel use states to be combined into one use state. Data flow analysis of the following statements

```
1.    A = X
2.    IF(A .GT. Y)
3.          A = Y
4.    ELSE
5.          Y = A
6.    END IF
```

would be performed in the following steps:

1. Statement 1 uses X as input and A as output.

2. Statement 2 adds use of Y and A as input to the use-state and stacks the current use-state.

3. Statement 3 uses A as output and Y as input (this is already described in the use-state for the path originating at Statement 2).

4. Statement 4 pops the use-state and then stacks the use-state of the former path.

5. Statement 5 adds use of Y as output to the use-state for the second path originating at Statement 2.

6. Statement 6 pops the use-state stack and causes the two current use-states to be combined into one. The combined use-state is use of A as output and input on all paths, use of X as input on all paths, use of Y as input on all paths, and as output on some paths.

Iterative constructs are processed once in the same sequential manner. As described earlier, references to functions or subroutines will have known variable use properties since the function or subroutine will already have been analyzed. When the analysis of a module is complete, use of local variables as input on some or all paths indicates a data flow anomaly, and use of all global variables is available to compare with data access assertions and to define the use properties of the module.

An algorithm to automatically perform data flow analysis in the manner just described has been implemented in the Software Quality Laboratory. The implementation of the algorithm distinguishes three types of output useage. They are:

SO	output on all paths
O	output on some paths
N	not used as output.

Similarly three types of input useage are distinguished; they are:

SI	input on all paths
I	input on some paths
N	not used as input.

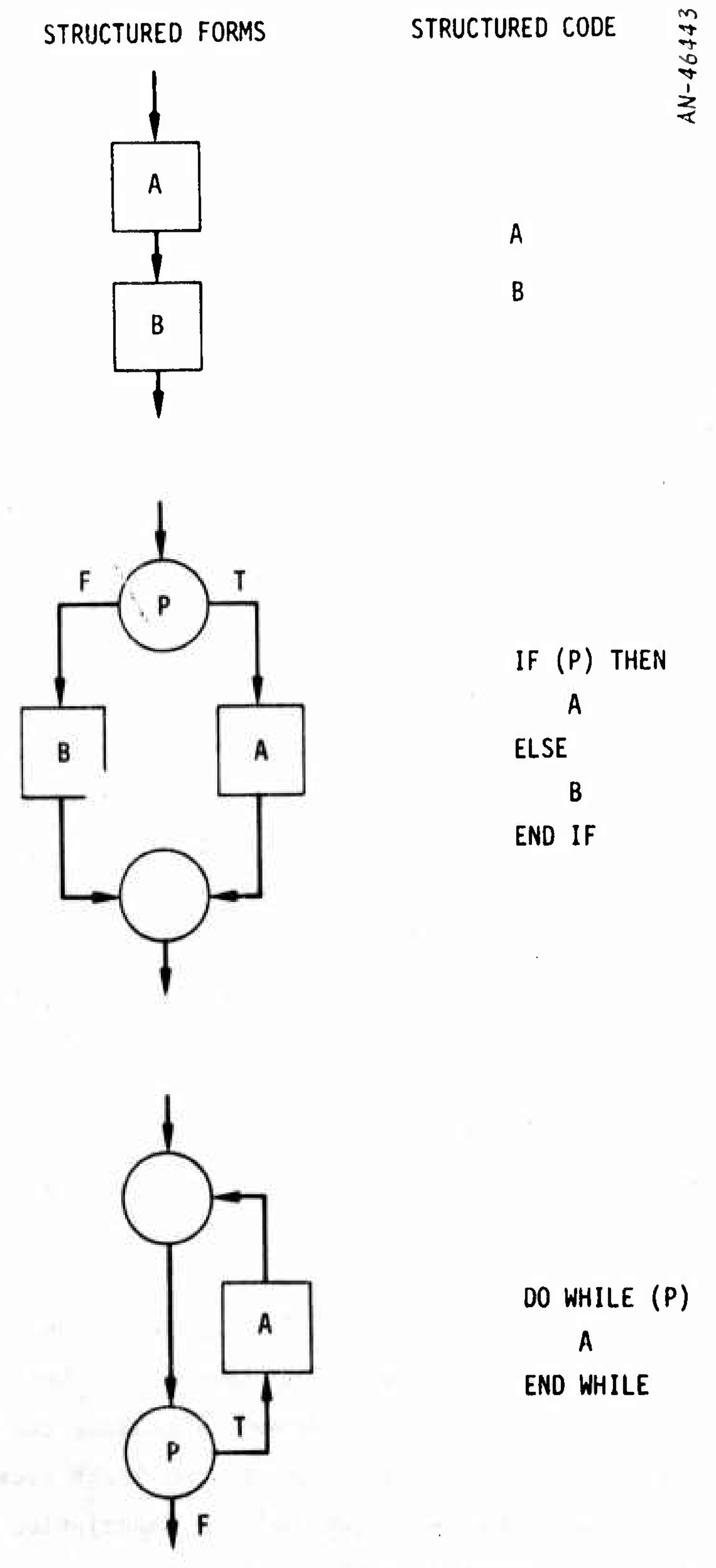
To allow for undefined externals, an additional unknown state, U , is distinguished. Ten input/ouptut states for a given variable are possible:

1	(N,N)
2	(SI,N)
3	(N,SO)
4	(SI,SO)
5	(I,N)
6	(N,O)
7	(I,O)
8	(I,SO)
9	(SI,O)
10	(U)

A local variable is used before being assigned a value if its input state is SI . It may be used before having a value if its input state is I . A global variable's actual use is consistent with asserted input usage only if its input state is SI or I , and consistent with asserted output usage only if its output state is SO or O . The use state for a path is implemented as a 2-by-N array, where N is the number of variables which have been used or set. The use of each variable is associated with its symbol table pointer, and the array is ordered by symbol table entry. A variable width stack is used to temporarily store use states during the data flow analysis of a module.

The data flow analysis algorithm can be applied to unstructured programs. A graphical analysis is performed which represents unstructured programs in a structured manner. The basic structured forms are sequential, parallel, and iterative combinations of single-entry, single-exit sequences of statements (Fig. 4.23). The graphical analysis performed is based on two properties of the three basic structured forms. Each of the basic forms can be characterized as a single entry/single exit subgraph. Also the graph of a structured program which uses only the basic forms is built up of well-nested single entry/single exit subgraph, the complexity of the graph can be reduced by replacing an identified form with a single edge which goes from its single entry to its single exit. It is then possible to repeat this process and identify basic forms which previously were composed of more complex structures than the edges which have been inserted. The process terminates when the resulting graph consists of a single edge from the entry of the module to the exit from the module. As the reductions are being performed, it is essential to maintain a data structure which indicates the basic form identified and the single entry/single exit (SE/SE) subgraphs of which it is composed. The natural data structure for this information is a hierarchy of SE/SE subgraphs. The structure of this heirarchy corresponds directly with the well-nested, indented representation of the text as a structured program.

The SE/SE tree (the structured graphical representation of the program) is used to perform the data flow analysis of both structured and unstructured programs. The algorithm involves walking the SE/SE tree and computing the use state for each node in the SE/SE tree after the use states of its sub-trees have been computed. A description of the data flow analysis algorithm is presented in Fig. 4.24.



```
A
B
```

```
IF (P) THEN
    A
ELSE
    B
END IF
```

```
DO WHILE (P)
    A
END WHILE
```

Figure 4.23. Basic Structured Forms

```
SEQ NEST SOURCE            KWOK

         CKWOK

                PROCEDURE DATA FLOW ANALYSIS
                WHILE SE/SE TREE WALK IS NOT FINISHED
     1          .    IF THE CURRENT NODE IS A TERMINAL NODE
     2          .    .    INVOKE( DD-PATH NODE )
     2          .    .    MOVE UP TO PARENT NODE
     1          .    ELSE
     2          .    .    IF DATA FLOW OF EACH SUBTREE HAS BEEN COMPUTED
     3          .    .    .    TYPE OF REDUCTION IS NAME OF NODE IN THE SE/SE TREE
     3          .    .    .    INVOKE( USE-SET MAPPING )
     3          .    .    .    MOVE UP TO PARENT NODE
     2          .    .    ELSE
     3          .    .    .    MOVE TO NEXT NODE IN LEFT TO RIGHT WALK
     2          .    .    END IF
     1          .    END IF
                END WHILE

                BLOCK( DD-PATH NODE )
     1          .    START WITH THE FIRST STATEMENT IN DD-PATH
     1          .    REPEAT
     2          .    .    IF AN EXECUTABLE STATEMENT
     3          .    .    .    INVOKE( ARRAY SUBSCRIPT USE-SET )
     3          .    .    .    INVOKE( EXTERNAL REFERENCE USE-SET )
     3          .    .    .    IF AN ASSSIGNMENT STATEMENT
     4          .    .    .    .    INVOKE ( ASSIGNMENT STATEMENT USE-SET )
     3          .    .    .    ORIF THIS IS A READ STATEMENT
     4          .    .    .    .    INVOKE( READ STATEMENT USE-SET )
     3          .    .    .    END IF
     3          .    .    .    INVOKE( DEFAULT USE-SET )
     2          .    .    ORIF THIS IS A PRESET STATEMENT
     3          .    .    .    INVOKE( PRESET STATEMENT USE-SET )
     2          .    .    END IF
     2          .    .    PUT USE-SET ON TOP OF STACK
     2          .    .    IF THIS IS NOT THE FIRST STATEMENT IN DD-PATH
     3          .    .    .    TYPE OF REDUCTUON IS DD-PATH
     3          .    .    .    INVOKE( USE-SET MAPPING )
     2          .    .    END IF
     2          .    .    USE NEXT STATEMENT IN DD-PATH
     1          .    UNTIL ALL STATEMENTS HAVE BEEN ANALYZED
                END BLOCK

                BLOCK( USE-SET MAPPING )
     1          .    IF TYPE OF REDUCTION IS NOT A SELFLOOP
     2          .    .    GET USE-SET ON TOP OF STACK
     2          .    .    GET USE-SET ON TOP OF STACK
     2          .    .    COMBINE USE-SETS ACCORDING TO REDUCTION TYPE
     2          .    .    PUT RESULTANT USE-SET ON TOP OF STACK
     1          .    ELSE
     2          .    .    GET USE-SET ON TOP OF STACK
     2          .    .    GET USE-SET ON TOP OF STACK
     2          .    .    SAVE LAST USE-SET ON TOP OF STACK
     2          .    .    COMBINE USE-SETS FOR A LINEAR COMBINATION
     2          .    .    GET USE-SET ON TOP OF STACK
     2          .    .    COMBINE USE-SETS FOR A PARLEL COMBINATION
     2          .    .    PUT RESULTANT USE-SET ON TOP OF STACK
     1          .    END IF
                END BLOCK
                END
```

Figure 4.24. Data Flow Analysis Algorithm

Several limitations are inherent in the simplicity of the data flow analysis technique. All structurally possible paths are included in the data flow analysis, even though many of them may not be logically executable. Also arrays are treated as single variables. The use of individual array elements is not distinguished. The implementation currently available in the Software Quality Laboratory does not attempt to recognize equivalenced variables during the data flow analysis. These limitations can cause invalid error and warning messages to be generated. They do not cause any data flow anomalies to go undetected however.

5 EXECUTION

Once the techniques described in Sec. 4 have been applied to show that many of the costly semantic errors are not present, the program is ready for an execution test. In the execution test, the Software Quality Laboratory provides facilities to show what paths have been tested, what paths have not been tested, what assertions are false, what the values of the input variables were on module entry, and what the values of the output variable were on module exit. These facilities may be enabled or disabled by the tester.

5.1 COVERAGE REPORTS

The reports which give information as to how well the software is being tested in terms of number of tested paths, numbers of untested paths, and number of times each path was executed are known as the QUICKLOOK coverage reports. They can aid the tester in devising additional test cases, in discovering paths which cannot be executed due to the range of valid data, and in pinpointing areas of the program in which most of the execution time is spent.

For the QUICKLOOK coverage reports, the modules for which data are to be collected must contain statements which are treated as comments by the preprocessor. Such modules are said to be instrumented.

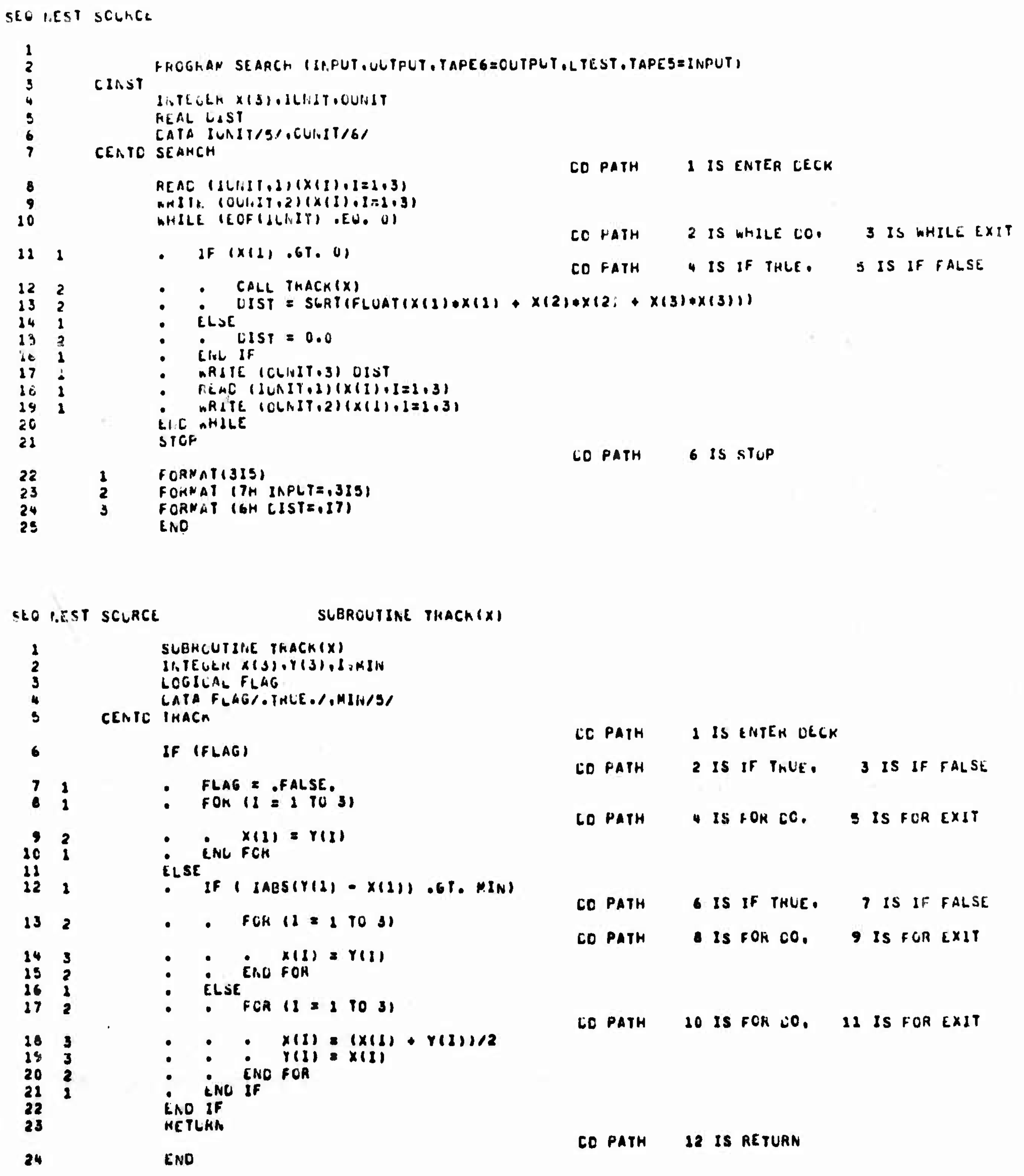
For IFTRAN the comment statement

```
CENTD name
```

where name is the module name is placed just before the first executable statement in the program. The accompanying figure, Fig. 5.1, shows the modules SEARCH and TRACK with these statements which define the modules' entry points.

In addition, one module, usually the first must have the comment statement

```
CINST
```



```
SEQ NEST SOURCE

  1
  2                 PROGRAM SEARCH (INPUT,OUTPUT,TAPE6=OUTPUT,LTEST,TAPE5=INPUT)
  3          CINST
  4                 INTEGER X(3),IUNIT,OUNIT
  5                 REAL DIST
  6                 DATA IUNIT/5/,OUNIT/6/
  7          CENTO SEARCH
                                                                  DD PATH     1 IS ENTER DECK
  8                 READ (IUNIT,1)(X(I),I=1,3)
  9                 WRITE (OUNIT,2)(X(I),I=1,3)
 10                 WHILE (EOF(IUNIT) .EQ. 0)
                                                                  DD PATH     2 IS WHILE DO,     3 IS WHILE EXIT
 11  1              .   IF (X(1) .GT. 0)
                                                                  DD PATH     4 IS IF TRUE,     5 IS IF FALSE
 12  2              .   .   CALL TRACK(X)
 13  2              .   .   DIST = SQRT(FLOAT(X(1)*X(1) + X(2)*X(2) + X(3)*X(3)))
 14  1              .   ELSE
 15  2              .   .   DIST = 0.0
 16  1              .   END IF
 17  1              .   WRITE (OUNIT,3) DIST
 18  1              .   READ (IUNIT,1)(X(I),I=1,3)
 19  1              .   WRITE (OUNIT,2)(X(I),I=1,3)
 20                 END WHILE
 21                 STOP
                                                                  DD PATH     6 IS STOP
 22          1      FORMAT(3I5)
 23          2      FORMAT (7H INPUT=,3I5)
 24          3      FORMAT (6H DIST=,I7)
 25                 END


SEQ NEST SOURCE                  SUBROUTINE TRACK(X)

  1                 SUBROUTINE TRACK(X)
  2                 INTEGER X(3),Y(3),I,MIN
  3                 LOGICAL FLAG
  4                 DATA FLAG/.TRUE./,MIN/5/
  5          CENTO TRACK
                                                                  DD PATH     1 IS ENTER DECK
  6                 IF (FLAG)
                                                                  DD PATH     2 IS IF TRUE,     3 IS IF FALSE
  7  1              .   FLAG = .FALSE.
  8  1              .   FOR (I = 1 TO 3)
                                                                  DD PATH     4 IS FOR DO,     5 IS FOR EXIT
  9  2              .   .   X(1) = Y(I)
 10  1              .   END FOR
 11                 ELSE
 12  1              .   IF ( IABS(Y(1) - X(1)) .GT. MIN)
                                                                  DD PATH     6 IS IF TRUE,     7 IS IF FALSE
 13  2              .   .   FOR (I = 1 TO 3)
                                                                  DD PATH     8 IS FOR DO,     9 IS FOR EXIT
 14  3              .   .   .   X(I) = Y(I)
 15  2              .   .   END FOR
 16  1              .   ELSE
 17  2              .   .   FOR (I = 1 TO 3)
                                                                  DD PATH    10 IS FOR DO,    11 IS FOR EXIT
 18  3              .   .   .   X(I) = (X(I) + Y(I))/2
 19  3              .   .   .   Y(I) = X(I)
 20  2              .   .   END FOR
 21  1              .   END IF
 22                 END IF
 23                 RETURN
                                                                  DD PATH    12 IS RETURN
 24                 END
```

Figure 5.1. Instrumented Modules

which turns on the instrumentation for the module in which the statement appears and all subsequent modules. Instrumentation may be turned off in selected modules by use of the statement

```
CNOIN
```

which disables the data collection process until another

```
CINST
```

is encountered.

Besides preparing the module itself, a QUICKLOOK command set is used to select reports which may be presented. This command set appears just after the data cards. If there is an end of file at the end of the data, it must be read by the module.

Three types of commands are used: the module select commands, the report select commands, and the print command.

The module select commands allow reports to be printed for specified modules. The two commands are:

```
QUICKLOOK MODULE name
QUICKLOOK DETAILED name
```

If the DETAILED report is desired, the DETAILED command must be used. The DETAILED report presents the information in a graphical form which is easy to read.

While the module select commands state the module names for which data are to be collected, the report select commands select which printouts are to be given. There are four print options: SUMMARY, NOTHIT, DETAILED, and CUMULATIVE.

The command

```
QUICKLOOK PRINT SUMMARY
```

gives the following information in tabular form covering all the selected modules:

1. Test case number
2. Module names and number of decision-to-decision paths
3. Number of module invocations, number of decision-to-decision paths traversed, percent coverage for this test case
4. Total number of module invocations, number of decision-to-decision paths, and percent coverage for all test cases.

Figure 5.2 shows the SUMMARY report as a result of executing the modules SEARCH and TRACK with the two sets of input data x = {5,3,6} and x = {7,2,7} .

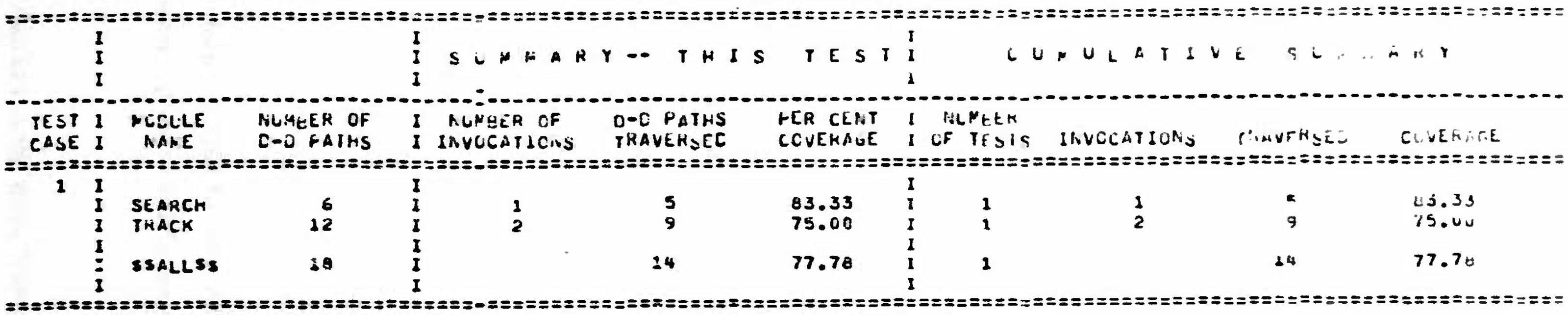
At this point the tester might decide further testing is necessary and look at the NOTHIT report to see what paths were not executed and refer back to the program listing to see what additional test cases could cause the paths to be executed.

The command

```
QUICKLOOK PRINT NOTHIT
```

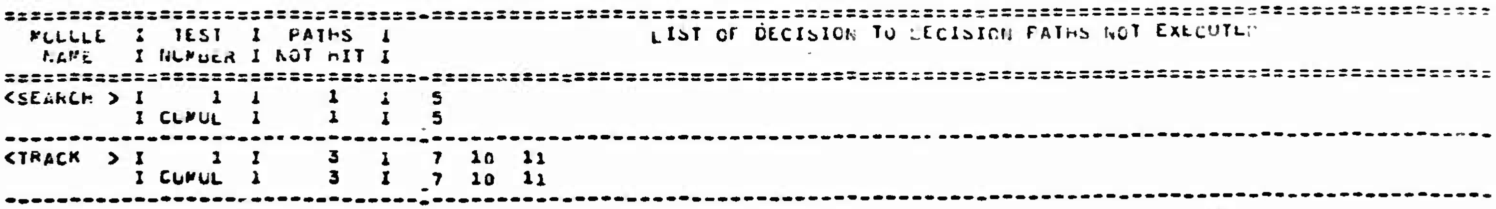
presents the following report (next page) listing the decision-to-decision path numbers for each module which were not executed. The paths are listed for each test case and for all test cases.

The tester might in addition look at the DETAILED and CUMULATIVE reports to see the unexecuted paths and note which paths are executed the most frequently. The DETAILED report shows for each test case the number of times a decision-to-decision path was executed in graphical form and gives overall coverage data for each module that was selected.



```
==================================================================================================================================
          I                              I                                          I
          I                              I  S U M M A R Y --  T H I S   T E S T     I       C U M U L A T I V E   S U M M A R Y
          I                              I                                          I
----------------------------------------------------------------------------------------------------------------------------------
 TEST  I  MODULE      NUMBER OF          I  NUMBER OF     D-D PATHS    PER CENT      I  NUMBER
 CASE  I   NAME       D-D PATHS          I  INVOCATIONS   TRAVERSED    COVERAGE      I  OF TESTS   INVOCATIONS   TRAVERSED   COVERAGE
==================================================================================================================================
   1   I                                 I                                          I
       I  SEARCH          6              I      1             5          83.33      I     1            1            5          83.33
       I  TRACK          12              I      2             9          75.00      I     1            2            9          75.00
       I                                 I                                          I
       I  $$ALL$$        18              I                   14          77.78      I     1                        14          77.78
       I                                 I                                          I
==================================================================================================================================
```

Figure 5.2. Summary Report for Test of Search and Track



```
==================================================================================================================
 MODULE    I  TEST    I  PATHS   I                     LIST OF DECISION TO DECISION PATHS NOT EXECUTED
  NAME     I  NUMBER  I  NOT HIT I
==================================================================================================================
<SEARCH >  I      1   I     1    I  5
           I  CUMUL   I     1    I  5
------------------------------------------------------------------------------------------------------------------
<TRACK  >  I      1   I     3    I  7  10  11
           I  CUMUL   I     3    I  7  10  11
------------------------------------------------------------------------------------------------------------------
```

NOTHIT Report

The CUMULATIVE command presents the results of several test cases in the same graphical format as in the DETAILED report. Figure 5.3 shows the DETAILED report for the test of the modules STACK and SEARCH. The CUMULATIVE report is the same as the DETAILED report in this case.

These reports are generated as a result of the two commands:

```
QUICKLOOK PRINT DETAILED
QUICKLOOK PRINT CUMULATIVE
```

The third type of command is

```
QUICKLOOK
```

which provides a listing of the requests that were made such as is shown in Fig. 5.3 (next page).

As an example of a QUICKLOOK command set which produced all the possible reports for a set of two modules, one of which is the main program SEARCH and the other which is the subprogram TRACK is:

```
QUICKLOOK DETAILED SEARCH
QUICKLOOK DETAILED TRACK
QUICKLOOK PRINT SUMMARY
QHICKLOOK PRINT NOTHIT
QHICKLOOK PRINT DETAILED
QUICKLOOK PRINT CUMULATIVE
QUICKLOOK
```

5.2 ASSERTIONS

Executable assertions may be used during execution to aid in testing and to aid in generating the correct assertions which can be used in the formal verification of a program.

During the testing phase, executable assertions are valuable in checking that interfaces have been specified correctly. While the static

```
RECORD OF DECISION TO DECISION (DD PATH) EXECUTION

-------------------------------------------------------------------------------------------------------------

MODULE   $SEARCH $          TEST CASE NO.        1

-------------------------------------------------------------------------------------------------------------
DD PATH I  NO.  NOT EXECUTED  I   NUMBER OF EXECUTIONS -- NORMALIZED TO MAXIMUM         I        I  NUMBER OF
NUMBER  I                     I.-------20.-------40.-------60.-------80.------100.      I        I EXECUTIONS
-------------------------------------------------------------------------------------------------------------
        I                     I
    1   I                     I xxxxxxxxxxxxxxxxxxxxxxxxx                               I    1   I        1
    2   I                     I xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx      I    2   I        2
    3   I                     I xxxxxxxxxxxxxxxxxxxxxxxxx                               I    3   I        1
    4   I                     I xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx      I    4   I        2
    5   I   5           00000 I                                                         I        I
    6   I                     I xxxxxxxxxxxxxxxxxxxxxxxxx                               I    6   I        1
-------------------------------------------------------------------------------------------------------------
                                                          TOTAL NUMBER OF DD PATH EXECUTIONS =        7

   TOTAL OF  1  NOT EXECUTED     EXECUTED   5/  6        PERCENT EXECUTED =    83.33
```

Figure 5.3. Detailed Reports

```
RECORD OF DECISION TO DECISION (DD PATH) EXECUTION
------------------------------------------------------------------------------------------------------------

MODULE  STRACK  $          TEST CASE NO.         1

------------------------------------------------------------------------------------------------------------
DD PATH I  NO.  NOT EXECUTED  I   NUMBER OF EXECUTIONS -- NORMALIZED TO MAXIMUM       I         I  NUMBER OF
NUMBER  I                     I.-------20.-------40.-------60.-------80.------100.   I         I EXECUTIONS
------------------------------------------------------------------------------------------------------------
        I                     I
   1    I                     I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX                     I    1    I          2
   2    I                     I XXXXXXXXXXXXXXXX                                      I    2    I          1
   3    I                     I XXXXXXXXXXXXXXXX                                      I    3    I          1
   4    I                     I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX    I    4    I          3
   5    I                     I XXXXXXXXXXXXXXXX                                      I    5    I          1
   6    I                     I XXXXXXXXXXXXXXXX                                      I    6    I          1
   7    I   7         00000   I                                                       I         I
   8    I                     I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX    I    8    I          3
   9    I                     I XXXXXXXXXXXXXXXX                                      I    9    I          1
  10    I  10         00000   I                                                       I         I
  11    I  11         00000   I                                                       I         I
  12    I                     I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX                     I   12    I          2
------------------------------------------------------------------------------------------------------------

                                                        TOTAL NUMBER OF DD PATH EXECUTIONS =      15

   TOTAL OF  3  NOT EXECUTED     EXECUTED   9/ 12       PERCENT EXECUTED =    75.00
```

Figure 5.3. Continued

```
COVERAGE DATA WILL BE ANALYZED FOR ALL ENCOUNTERED INSTRUMENTED MODULES

DETAILED ANALYSIS WILL RESULT FOR THE FOLLOWING MODULES

            -----------
            MODULE NAME
            -----------
            SEARCH
            TRACK

      REPORT          I    ON/OFF FLAG
  --------------------I----------------
   TEST RESULT        I        T
   CUMMULATIVE        I        T
   SUMMARY            I        T
   NOT HIT            I        T
```

Report from Command QUICKLOOK

analysis techniques described in Sec. 4 will detect most of the serious errors, there still remains the possibility of data being out of the expected range of values. This kind of error can be detected by explicitly stating range restrictions on every input variable in the INITIAL assertion. If an error occurs the name of the module and the assertion number is reported on a listing. A trace of the values of the input variable values which is given by the INPUT assertion should then aid in the detection of the source of the error. The fact that in one module the range of the values generated is not consistent with what is expected by a subsequent module will be immediately obvious.

Thus if processing data from a radar which provides a beam number between 2 and 1021, a range bin number between 80 and 1600, and a signal strength number between 0 and 95, one should have in the processing routine which receives the data as the first executable statements:

```
 INPUT (/INTEGER/BEAM,RANGE,SIGNAL)
 INITIAL (2 .LE. BEAM .AND. BEAM .LE. 1021)
*        .AND. 80 .LE. RANGE .AND. RANGE .LE. 1600
*        .AND. 0 .LE. SIGNAL .AND. SIGNAL .LE. 95)
```

for IFTRAN

```
INPUT beam, range, signal;
INITIAL 2 <= beam AND beam <= 1021
        AND 80 <= range AND range <= 1600
        AND 0 <= signal AND signal <= 95;
```

for V-PASCAL.

Similar processing is available for the output variables which can be checked for their range of values. If for the example the radar data described above was transformed into a cartesian coordinate system where the X,Y,Z coordinates were in terms of meters, and it was known that X could be between 500 and 100,000 meters, Y could be between -50,000 and 40,000 meters, and Z could be between 50 and 75,000 meters, a reasonableness check could be:

```
OUTPUT (/REAL/X,Y,Z)
FINAL (500.0 .LE. X .AND. X .LE. 100000.0 .AND.
     -50000 .LE. Y .AND. Y .LE. 40000.0 .AND.
     50.0 .LE. Z .AND. Z .LE. 75000.0)
```

A better check would be to relate the outputs: X,Y and Z to the inputs such as making use of the relation between range and the outputs:

If

$$\text{range}^2 = X^2 + Y^2 + Z^2$$

and if

radar range * k = range in meters

we could state

```
FINAL ((RANGE*K)**2 - (X**2 + Y**2 +Z**2) .LE. RNDOFF)
```

for IFTRAN and

```
FINAL (range*k)*(range*k) - (x*x + y*y + z*z) <= rndoff;
```

for V-PASCAL.

5.3 FAULT DETECTION

Once a program has been verified, it is known that it will produce the specified results as stated in the FINAL assertion under the conditions that are stated in the INITIAL assertion. The assertions can then be used for detection of faults due to bad input data or bad hardware.

One of the possible causes of catastrophic failure in a computer system is bad input data. If the data is from a sensor that normally provides more data than is necessary, the bad value can be discarded and the system can proceed to accept another value until n bad values in a row.

The assertions with a FAIL clause can be used to provide for fault detection with a user-supplied block that can be used for fault recovery.

If, for example, it was known that the input SIGNAL should fall between 5 and 95, and if SIGNAL is invalid an error flag should be set, the following assertion could be used:

```
 ERROR = .FALSE.
 ASSERT (SIGNAL .GE. 5 .AND. SIGNAL .LE. 95)
*      FAIL (ERROR FIX)
 IF (.NOT. ERROR)
       normal processing
 END IF
 .
 .
 .
 BLOCK (ERROR FIX)
      ERROR = .TRUE.
 END BLOCK
 .
 .
 .
```

In this case, if the input did not meet the specification, rather than a report on the listirg of a false assertion, the user-supplied block named ERROR FIX is invoked to set the ERROR flag.

In V-PASCAL, the same assertion would be:

```
error := false;
ASSERT signal >= 5 AND signal <= 95
FAIL error := true END FAIL;
IF NOT error THEN
      normal processing
END IF;
.
.
.
```

Rather than skip the processing the designer might decide to set SIGNAL to a nominal value or to an old value.

This could be done by

```
    ASSERT (SIGNAL .GE. 5 .AND. SIGNAL .LE. 95)
   *      FAIL (ERROR FIX)
    normal processing
    .
    .
    .
    BLOCK (ERROR FIX)
    SIGNAL = NOMINL
    END BLOCK
```

While the same results could be obtained by a series of IF tests in either language, the advantages of using the assertions are:

1. The conditions under which the code that follows is expected to operate are explicitly stated

2. The methods of handling errors due to input data are separated from the rest of the code

3. The assertions used in fault detection are the ones used in a formal verification

6 FORMAL VERIFICATION

A major part of the work is the development of a capability for the verification of FORTRAN and PASCAL. This has resulted in the design and implementation of a verification condition generator, a simplifier, and an interactive simplifier which are able to verify single modules.

6.1 VERIFICATION TOOLS

6.1.1 Design and Implementation of VCG

The verification condition generator produces verification conditions for programs with assertions written in FORTRAN, IFTRAN, or the subset of PASCAL limited to FORTRAN-like data types.

The verification condition generator uses assertions which have been inserted into the source code to generate verification conditions in the form $A \rightarrow B$, where A is the initial assertion on a program path conjuncted with the predicates encountered along the path and B is the assertion at the end of the path. All variables in the verification condition are represented in terms of their symbolic value at the start of the path. The required substitutions are made by symbolically executing the final assertion and any predicates backwards to the initial assertion.

Three keywords are used to state assertions in the present system. These are INITIAL, FINAL, and ASSERT. The INITIAL assertion is a statement of the conditions which are true when the module is entered. The FINAL assertion is a statement of the conditions that are true on exit from the module. The ASSERT statement is normally used to express loop invariants, but may be used anywhere in the body of the module to express a condition that is true at that point. The syntax of the assertions is discussed in Sec. 3. First order predicate calculus statements about program variables may be expressed in the assertions.

Program structure is used to determine the set of verification conditions to generate. A well-structured, single-entry, single-exit program has a readily identified set of verification conditions. There is a verification condition for each verification path in the program. Verification paths begin at the program entry point and loop entry points, and end at the program exit point and loop exit points. Each logically possible path between program entry, loop entry, loop exit, and program exit corresponds to a verification path. Each verification path must begin and end with an INITIAL, FINAL, or ASSERT statement.

Specification of verification conditions is presently handled by stating the set of DD-paths which lie between assertions. For example, a verification condition is generated by giving the static analysis system a command of the form:

```
VCG, PATH = 2,1,3.
```

"VCG" commands the system to invoke the verification condition generator. PATH = 2,1,3 is an example of the present method of specifying the path over which the condition is to be generated. The general form of the command is

```
VCG,PATH = <no. of paths>{,<dd path number>}
```

In the example, two decision-to-decision (dd) paths are specified: path number 1 and path number 3. The verification condition generator will take the first assertion on path 1, the last assertion on path 3, and the intervening body of code to generate a verification condition.

The program SIMP is a very simple IFTRAN program which contains three DD-paths.

```
PROGRAM SIMP (INPUT, OUTPUT)
ENTRY (.TRUE.)
A = 5.0
B = 0.0
WHILE (A .GE. 0.0)
     ASSERT (A .GE. 0.0 .AND. B .LE. A**(-2))
     B = A**(-2)
```

```
END WHILE
EXIT (B .LE. 1.OE-14)
STOP
END
```

Path 1 covers the program entry statement up to the WHILE statement. Path 2 covers the "true" part of the WHILE statement and the statements in the loop. Path 3 covers the "false" part of the WHILE statement and the statements following the END WHILE.

The verification condition generator, when given the command VCG, PATH = 2,1,3 will take the assertion on path 1, ENTRY (.TRUE.); the statements A = 5.0 and B = 0.0; the "false" of WHILE (A .GE. 0.0); and the assertion on path 3, EXIT (B .LE. 1.OE-14), to generate the verification condition:

$$(.TRUE.) \wedge (5.0\ .LE.\ 0.0) \rightarrow (0.0\ .LE.\ 1.OE\text{-}14)$$

In this simple example, the premise is false and hence the resulting condition is true. When the premise is false, the implication is that the selected path combination is impossible.

Although this method of selecting verification conditions to be generated provides flexibility for symbolic execution, a more automatic selection mechanism which relates to the assertions is under development.

Other approaches to verification condition generation are possible. Most existing program proving systems generate one complicated verification condition for an entire program. The advantages of associating verification conditions with verification paths are:

1. Verification conditions are smaller and more susceptible to automatic simplification

2. Verification conditions are more readable, allowing more intelligent interactive simplification

3. The size of verification conditions is independent of program size

4. Incorrect assertions or code are easily found since each verification condition is associated with a single verification path.

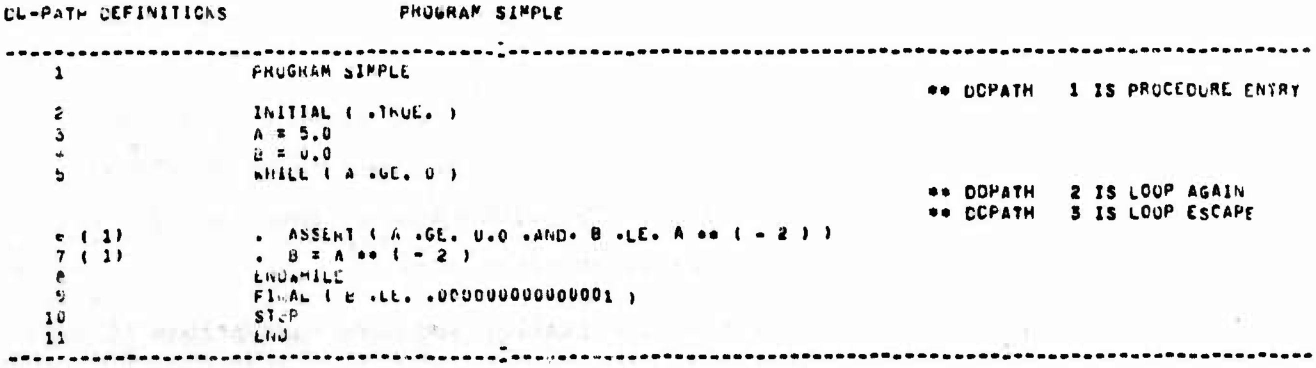
A potential disadvantage is the large number of verification conditions which may be required for some programs. This has not been a problem yet.

Several automatically produced reports are related to verification condition generation. Figure 6.1 shows the DD-path definitions report for program SIMPLE. This report provides the DD-path numbers used in specifying verification paths. It is produced with the commands:

```
MODULE = (SIMPLE).
PRINT, DDPATHS.
```

There is one verification path around the loop in SIMPLE. it is specified with the command

```
VCG,PATH = 2,2,2.
```



```
DD-PATH DEFINITIONS                 PROGRAM SIMPLE

-----------------------------------------:-----------------------------------------------------------------
    1                   PROGRAM SIMPLE
                                                                   ** DDPATH   1 IS PROCEDURE ENTRY
    2                   INITIAL ( .TRUE. )
    3                   A = 5.0
    4                   B = 0.0
    5                   WHILE ( A .GE. 0 )
                                                                   ** DDPATH   2 IS LOOP AGAIN
                                                                   ** DDPATH   3 IS LOOP ESCAPE
    6 ( 1)              .  ASSERT ( A .GE. 0.0 .AND. B .LE. A ** ( - 2 ) )
    7 ( 1)              .  B = A ** ( - 2 )
    8                   ENDWHILE
    9                   FINAL ( B .LE. .000000000000001 )
   10                   STOP
   11                   END
-----------------------------------------:-----------------------------------------------------------------
```

Figure 6.1. DD-Path Definitions for Module SIMPLE

Figure 6.2 is produced as a result of this command. It identifies the statements in the specified verification path. This verification path corresponds to going around the loop in SIMPLE once. Figure 6.3 is also produced as a result of the VCG,PATH command. It is the verification condition associated with the verification path in Fig. 6.2. Each term in the verification condition is directly related to the source line number from which it was derived.

```
LINE                PATH SOURCE TEXT

   5                WHILE ( A .GE. 0 )
   6 ( 1)           .  ASSERT ( A .GE. 0.0 .AND. B .LE. A ** ( - 2 ) )
   7 ( 1)           .  B = A ** ( - 2 )
   8                ENDWHILE
   5                WHILE ( A .GE. 0 )
   6 ( 1)           .  ASSERT ( A .GE. 0.0 .AND. B .LE. A ** ( - 2 ) )
```

Figure 6.2. Verification Path Report for Module SIMPLE

```
             LINE     VERIFICATION CONDITION

                5     A .GE. 0

                       AND

                6     A .GE. 0.0 .AND. B .LE. A ** - 2

                       AND

                5     A .GE. 0
       ----------- IMPLIES -----------------------------

                6     A .GE. 0.0 .AND. A ** - 2 .LE. A ** - 2
```

Figure 6.3. Verification Condition Report for Module SIMPLE

All variables in the verification condition are represented in terms of their symbolic value at the start of the verification path. The verification condition is constructed as the verification path is symbolically executed in reverse order. Each statement type encountered when traversing the verification path in reverse order affects the verification condition being generated. The rules for FORTRAN are:

- When a decision statement or assertion is encountered, the appropriate condition is added as an .AND. term to the current formula (the final assertion is added as the consequence of the .IMP.).
- An assignment statement of the form $x = y$ causes all instances of the term x to be replaced with y in the current formula.
- An iteration control statement, such as the statement at the end of a FORTRAN DO-loop, causes all instances of the iteration index to be replaced with the incremented value. This is an assignment statement of the form: <index> = <index> + <increment>.
- An iteration initiation statement, such as a FORTRAN DO-statement, causes replacement of instances of the <index> with its <initial-value>.
- A statement label assignment results in replacement of instances of the label-name with the actual label.

Planned extensions to the verification condition generator will allow subroutine, function, and READ statements to be symbolically executed when the corresponding subroutine, function, or I/O unit has been defined using INITIAL, ASSERT, and FINAL statements.

6.1.2 Simplifier Design and Implementation

The verification condition simplifier consists of two separate parts: a standard simplifier and a user supplied simplifier. The standard simplifier applies a small set of arithmetic, logical, and relational simplifications in an attempt to reduce the verification condition to "true." The result of this attempt is presented to the user who can then supply additional simplification rules, which are peculiar to the problem at hand. Once a new rule has been applied, the modified result is sent through the simplifier again and the new result is presented to the user. In this manner, the user can verify the programs that the standard simplifier cannot.

6.1.2.1 Standard Simplification

The simplifier first puts the verification condition into a tree data structure. The root of the tree contains the implies operation. In Fig. 6.4, the tree for the expression

$$A > 2 \rightarrow A > 2 \vee A = 2$$

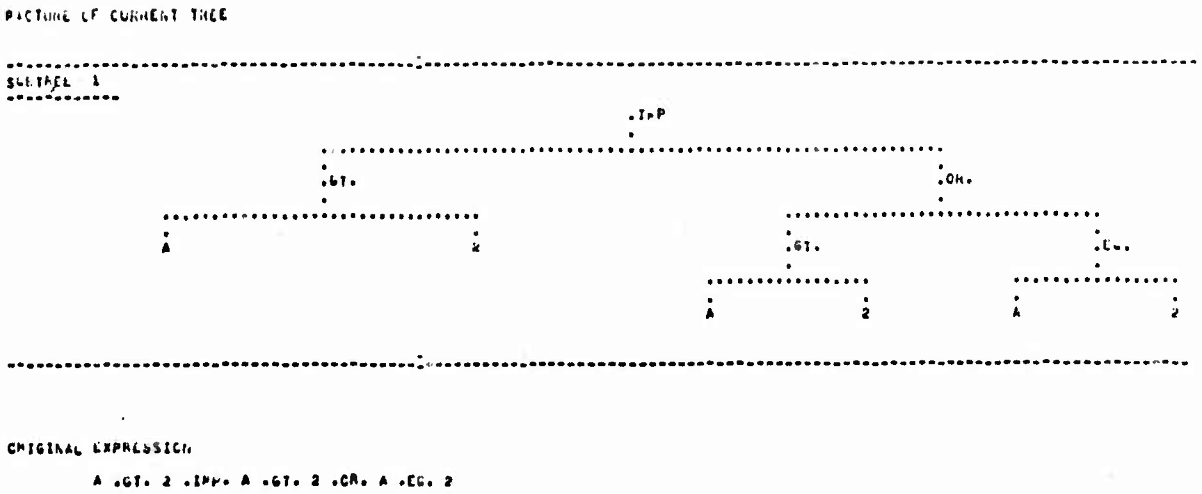
is shown as it is seen by the simplifier. A small set of tree operations were defined so that it was not difficult in IFTRAN to build trees, walk trees, delete nodes, move nodes, or print trees in the form shown in the figure.

Once the tree has been formed, a lexical level is assigned to each leaf so that a lexical ordering of the nodes can be performed. This allows the simplifier to recognize that the expressions

$$\begin{gathered} A + B + C \\ A + C + B \\ B + A + C \\ B + C + A \\ C + A + B \\ C + B + A \end{gathered}$$

are all the same. The lexical ordering would result in the preceding expression being replaced by

$$A + B + C$$



```
PICTURE OF CURRENT TREE

SUBTREE  1

ORIGINAL EXPRESSION
        A .GT. 2 .IMP. A .GT. 2 .OR. A .EQ. 2
```

Figure 6.4. Simplifier Tree

Constants have the lowest lexical order so any one of the expressions

A + 1 + B

1 + A + B

B + 1 + A

B + A + 1

1 + B + A

A + B + 1

would be replaced by

1 + A + B

Lexical ordering is part of the normalization process which takes place before the actual simplification takes place. Normalization is divided into four parts:

1. Products normalization
2. Conjunctive normalization

3. Relation normalization

4. Lexical normalization

Products normalization places expressions in a sum of products form. That is if the expression is

$$A * (B - C)$$

it will be normalized to

$$A * B - A * C$$

Products normalization also moves negations inwards to the individual terms that are being operated on. Double negations are removed. Some examples are:

Original	Normalized
$A - (A + B)$	$A - A + B$
$A - (-A + B)$	$A + A + B$
$-(-(A))$	A

Conjunctive normalization is similar to products normalization. The logical expressions are placed in conjunctive normal form on each side of the implies.

Original	Normalized
$(A \wedge B) \vee C$	$(A \vee C) \wedge (B \vee C)$
$(A \wedge B) \vee C \rightarrow (A \wedge D) \vee C$	$(A \vee C) \wedge (B \vee C) \rightarrow (A \vee C) \wedge (D \vee C)$

Conjunctive normalization is not applied across the implies because it is expected that human interaction will operate better when the verification condition simplifications are recognizable, when the clauses remain tied to the code and the assertions, and when the assertion on the right hand side of the implies is kept as a separate set of clauses. This is considered particularly important when a user attempts to generate a loop invariant from the verification condition as described in Sec. 6.1.3.

Just as products normalization brings in the negation operation next to individual terms, conjunctive normalization brings in the NOT operation next to individual terms. Double NOTs are removed.

Original	Normalized
$\neg\neg A$	A
$\neg(A \vee B)$	$\neg A \wedge \neg B$
$\neg(A \wedge \neg B)$	$\neg A \vee B$

Relational operators, the divide operator, and the exponential operator pose special problems. This simplifier leaves the division operator and the exponential operator as they appear in the original expression. Normalization across the relational operators takes place so that the normal form is

<variable expression> <relation> <constant>

Examples are:

Original	Normalized
$A > B$	$A - B > 0$
$3 < C$	$C < 3$
$A + 2 > D + 7$	$A - D > 7 - 2$

After a verification condition has been placed in normal form, it is simplified. The simplifier consists of five parts which are applied sequentially to each subtree during a post order walk of the tree. The five parts are:

1. Constant simplification
2. Common term simplification

3. Logical simplification
4. Cancellation
5. Relational simplification

It is assumed that the normalization process will have brought constant terms together. The constant simplification process will then evaluate arithmetic and logical expressions which contain constants. Special rules for 0 and 1 are incorporated in the constant simplification process. Constant expressions which are simplified may be real, integer, or logical data types. Examples of constant simplification are:

Original	Simplified
A > 3 + 6	A > 9
.TRUE. → INPUT3 > 0	INPUT3 > 0
.FALSE. → INPUT2 < 0	.TRUE.
B < 6 → .FALSE.	B ≥ 6
1 * RANGE + 6	RANGE + 6
0/TIME	0
0 + DIST	DIST
1 ** FINALV	1
.TRUE. ∧ (C > 6)	C > 6
1 + 2	3
6.0 + 3.0	9.0
2 < 7	.TRUE.
.TRUE. ∨ (F = MA)	.TRUE.

Common term simplification searches arithmetic expressions for equal terms which can be combined into a similar expression.

Original	Simplified
A - A + B	B
A * B - A * B	0
A * B + A * B	2 * A * B

Logical simplification does the same as common term simplification for the logical operators.

Original	Simplified
$(A + B > 5) \wedge (A + B > 5)$	$(A + B > 5)$
$(C < E) \vee (C < E)$	$(C < E)$

Cancellation of like terms across the implies is a separate part of the simplification process. If a clause on the right part of the implies is the same as a clause on the left hand side, the right hand clause is replaced by .TRUE.. The goal of the simplification process is to delete as many clauses as possible so that eventually the verification condition appears as:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_3 \rightarrow .TRUE.$$

which is simplified to .TRUE.. Examples of cancellation are:

Original Expression

$(A > 5) \wedge SORTED \wedge (L < N) \rightarrow SORTED$

Simplified Expression

.TRUE.

Original Expression

$$(\text{RANGE} > 0) \wedge (\text{ELEVATION} > 50) \rightarrow (\text{RANGE} > 0) \wedge (\text{AZIMUTH} > 3)$$

Simplified Expression

$$(\text{RANGE} > 0) \wedge (\text{ELEVATION} > 50) \rightarrow (\text{AZIMUTH} > 3)$$

Relational simplification takes conjuncted relations which involve equivalent terms and replaces them with the stronger relation.

Original	Simplified
$(\text{RANGE} > 3) \wedge (\text{RANGE} > 5)$	$(\text{RANGE} > 5)$
$(\text{TIME} \geq 6) \wedge (\text{TIME} > 6)$	$(\text{TIME} > 6)$
$(\text{DIST} > 3) \wedge (\text{DIST} < 2)$	.FALSE.
$(\text{SPEED} \geq 0) \vee (\text{SPEED} < 0)$	.TRUE.
$(\text{VEL} \geq 2) \wedge (\text{VEL} \leq 2)$	$(\text{VEL} = 2)$

If as a result of simplification a variable is equal to a constant, that constant replaces the variable in other clauses and the result is resimplified

Original

$$(B \geq 0) \wedge (B \leq 0) \rightarrow 1 = A{*}{*}B$$

Simplified

$$(B = 0) \rightarrow 1 = A{*}{*}B$$

Resimplified

.TRUE.

6.1.2.2 User-Supplied Simplifications

Although the standard simplifier contains many rules, it cannot automatically verify all the verification conditions from many programs.

Rather than change the simplifier or develop a complete theorem prover, the capability for adding rules to the simplification process was provided.

Two forms of rules are available. The first uses simple text replacement and the second uses pattern matching. Under text replacement, if a verification condition contained an expression of the form

$$B \geq 0$$

and the user desired to change this to

$$B > 0 \vee B = 0$$

the command sequence would be

```
VCG,REPLACE.
B .GE. 0 = B .GT. 0 .OR. B .EQ. 0
*END.
```

for IFTRAN OR

```
VCG,REPLACE.
B >= 0 = B > 0 OR B = 0
*END.
```

for V-PASCAL.

Such a command will cause the verification condition to be searched for the text string $B \geq 0$ which will be replaced with the text string $B > 0 \vee B = 0$. Then the standard simplifier will be reinvoked to see if the modified verification can be reduced to .TRUE. by the standard simplifier.

The replacement operation is implemented by the formation of a tree with the replacement equality operator as the root. The verification condition is searched for subtrees which are equal to the left subtree of the replacement tree. If found the right subtree is used for the replacements.

The more general method is to use pattern replacement rather than text strings which require exact matches. Pattern replacements are done with special pattern variables:

PX1,PX2,...,PX10.

An example of a pattern variable rule is:

$$\mathrm{PX1} > \mathrm{PX2} \wedge \mathrm{PX2} > \mathrm{PX3} = \mathrm{PX1} > \mathrm{PX3}$$

If this were applied to a verification condition which was

$$(\mathrm{RANGE} > \mathrm{MINRANGE}) \wedge (\mathrm{MINRANGE} > \mathrm{INPUT2}) \rightarrow (\mathrm{RANGE} > \mathrm{INPUT2})$$

PX1 would match RANGE

PX2 would match MINRANGE

PX3 would match INPUT2

so the replacement would result in

$$\mathrm{RANGE} > \mathrm{INPUT2} \rightarrow \mathrm{RANGE} > \mathrm{INPUT2}$$

which the simplifier would recognize as being .TRUE..

Pattern replacement rules are entered in the same manner as text replacement rules.

The preceding rule would be entered as

```
VCG,REPLACE.
PX1 .GE. PX2 .AND. PX2 .GT. PX3 = PX1 .GT. PX3
*END.
```

for IFTRAN, or as

```
VCG,REPLACE.
PX1 >= PX2 AND PX2 > PX3 = PX1 > PX3
*END.
```

for V-PASCAL.

By the combination of using the standard simplifier and the user supplied simplifier, several small programs have been formally verified as described in Sec. 6.3. In the process of verifying the programs, it was discovered that once a new rule was defined to verify part of a program, it was used repeatedly in the verification of that program. In order to save the effort of re-entering rules, the AXIOM command was implemented. Instead of giving the command sequence

```
VCG,REPLACE.
<rule>
*END.
```

one states

```
VCG,AXIOM.
<rule>
*END.
```

The rule will be assigned an axiom number and saved on a library of rules. A rule which does not result in any replacements will not be saved. Once on the library, the user need only refer to the axiom number as for example,

```
VCG,AXIOM,1.
VCG,AXIOM,3.
```

which would cause axiom 1 to be applied and then axiom 3.

6.1.3 Adding Assertions Using Verification Conditions

One of the methods proposed by Webgreit[22] to synthesize loop invariants uses the FINAL assertion and the exit condition from the loop. The trail loop invariant which is formed is then modified using a set of heuristics until it satisfies the conditions of a loop invariant. The Software Quality Laboratory provides a means whereby trail assertions may be placed in a program. The verification conditions which are generated from the trail assertions may be examined to see how to alter the assertion so that the verification conditions are valid.

By way of example, the text of the DIV subroutine which corresponds to Wegbreit's first example is used as shown in Fig. 6.5. Assuming that the INITIAL and FINAL assertions are provided, the problem is to find the loop invariant which is placed in an ASSERT statement. The ASSERT statement is expected to satisfy the following verification conditions for the loop:

1. Loop entry

   INITIAL $\wedge$ (loop entry)' $\rightarrow$ ASSERT'

2. Around the loop

   ASSERT $\wedge$ (loop continue)' $\rightarrow$ ASSERT'

3. Loop exit

   ASSERT $\wedge$ (loop exit)' $\rightarrow$ FINAL'

where

loop entry is the condition or predicate that causes entry to the loop

loop continue is the condition under which control remains in the loop

loop exit is the condition that causes transfer out of the loop

and INITIAL, ASSERT, and FINAL refer to the logical expressions in the assertions.

The primed terms refer to the logical expressions as they appear in terms of the variables that exist at the start of the loop.

For the example shown,

INITIAL = $A \geq 0 \wedge B \geq 0$

FINAL = $A = Q * B + R \wedge 0 \leq R \wedge R < B$

loop entry = loop continue $\equiv R \geq B$

loop exit = $R < B$

```
NO. LEVEL     LABEL   STATEMENT TEXT...                                                                           DDPATHS
------------------------------------------------------------------------------------------------------------------------
   1                  SUBROUTINE DIV ( A, B, Q, R )                                                               (  1)
   2          C       FCUN DIV
   3          C       KING S EXAMPLE 2
   4          C
   5          C       USE SUBTRACTION TO PERFORM DIVISION OF A BY B
   6          C       THE QUOTIENT IS IN Q WITH THE REMAINDER IN R
   7          C
   8                  INTEGER A, B, Q, R
   9          C
  10                  INITIAL ( A .GE. 0 .AND. B .GE. 0 )
  11                  Q = 0
  12                  R = A
  13                  WHILE ( R .GE. B )                                                                          (  2-  3)
  14 ( 1)             .  ASSERT ( .TRUE. )
  15 ( 1)             .  Q = Q + 1
  16 ( 1)             .  R = R - B
  17                  ENDWHILE
  18                  FINAL ( A .EQ. Q * B + R .AND. 0 .LE. R .AND. R .LT. B )
  19                  RETURN
  20                  END
------------------------------------------------------------------------------------------------------------------------


CLAUSE     VERIFICATION CONDITION

    1      - B + R .LT. 0 .OR. - ( 2 * B ) + R .GE. 0 .OR. A - R - ( B * Q ) .EQ.
            0
```

Figure 6.5. DIV Subroutine with Trail Loop Invariant

Wegbreit states that the standard means for generating a loop invariant is to start with the loop exit verification condition and use a trail verification condition

(loop exit)' → FINAL'

In the Software Quality Laboratory, a trail loop invariant can be generated by setting the loop invariant to .TRUE. as shown in the text of the program. A verification condition is then generated for the path around the loop. The resulting verification condition is then placed in disjunctive form by using the REPLACE command with the rule

PX1 .IMP. PX2 = .NOT. PX1 .OR. PX2

to remove the implies. The result of this operation is a trail loop invariant which is shown below the text of the program.

Now the three verification conditions are regenerated using the trail loop invariant. It is seen that the second verification condition cannot be reduced to true (see Fig. 6.6). One of the heuristics is to strengthen the assertion by changing the disjunction to a conjunction which will allow the second verification condition to be valid. When this is done, all the paths in the program can be verified using the standard simplifier as shown in the second verified program in Appendix D.

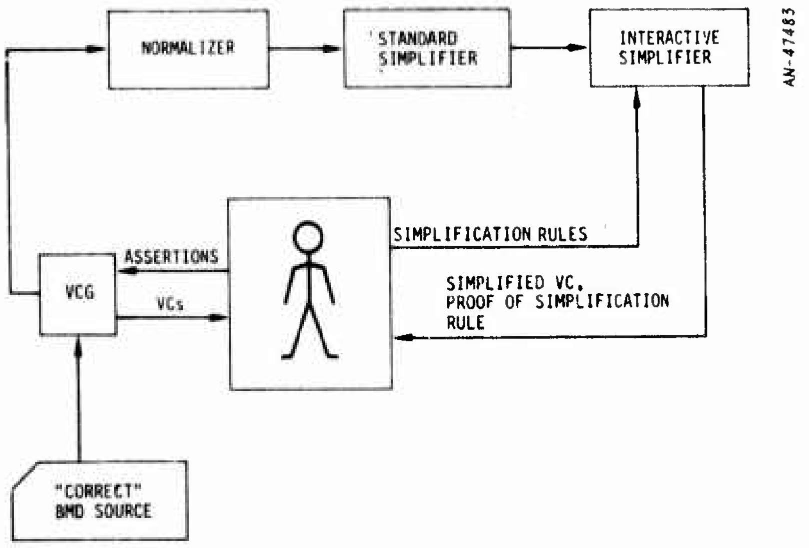
6.2 INTERACTIVE ASSISTANCE

An interactive interface to the Software Quality Laboratory has been implemented to aid the user in the development of assertions and improve the performance of the simplifier. Through the interface, the Software Quality Laboratory user can enter commands and receive output through the Anagraph.[20] A functional description of the relationship between the user and the Software Quality Laboratory is shown in Fig. 6.7.

Through the Anagraph terminal, the user can request verification conditions, provide trial assertions, specify additional simplification

```
VCG, PATH=2,2,2                        SUBROUTINE DIV ( A, B, Q, R )

LINE LABEL SOURCE TEXT

 13        WHILE ( R .GE. B )
 14        .ASSERT ( - B + R .LT. 0 .OR. - ( 2 * B ) + R .GE. 0 .OR. A - R - ( B
          ** Q ) .EQ. 0 )
 15        .Q = Q + 1
 16        .R = R - B
 17        ENDWHILE
 13        WHILE ( R .GE. B )
 14        .ASSERT ( - B + R .LT. 0 .OR. - ( 2 * B ) + R .GE. 0 .OR. A - R - ( B
          ** Q ) .EQ. 0 )

VCG, PATH=2,2,2                        SUBROUTINE DIV ( A, B, Q, R )

CLAUSE    VERIFICATION CONDITION

     1    - B + R .GE. 0

           AND

     2    - ( 2 * B ) + R .GE. 0

           AND

     3    ( - B + R .LT. 0 .OR. - ( 2 * B ) + R .GE. 0 .OR. A - R - ( B * Q ) .E
          Q. 0 )

----------- IMPLIES --------------------------------------------------

     4    - ( 2 * B ) + R .LT. 0 .OR. - B - ( 2 * B ) + R .GE. 0 .OR. A - R - (
          B * Q ) .EQ. 0
```

Figure 6.6. Loop Verification Condition Using Trail Loop Invariant



- USE PROGRAM STRUCTURE TO SIMPLIFY PROOF STEPS
- RELATE VERIFICATION CONDITIONS TO ORIGINAL PROGRAM

Figure 6.7. Interactive Program Proving

rules, and request the symbolic execution of expressions. In response, the Software Quality Laboratory generates verification conditions from assertions in the source code or assertions entered from the Anagraph, simplifies these verification conditions, symbolically executes arbitrary expressions over specified program paths, validates simplification rules, and applies them to verification conditions.

The reason for implementing the interactive simplification capability is to overcome a problem recognized by Deutsch:[6]

> There are two respects in which PIVOT has failed to attain the goals set for it by the author. One is stability. Even though PIVOT is restricted to a fairly limited domain, each new test case has required adding simplification rules or extending the logic of PIVOT in some way.

It is impractical to extend the capability of a simplifier to prove every possible program by including every possible simplification rule. It is more feasible to allow a user to add a rule that applies only to a single program.

The reason for giving the user the capability to interactively specify assertions and generate verification conditions is to assist the synthesis of loop invariants. Until loop invariants can be generated automatically, the user will need to supply these assertions. Since the specification of loop invariants is an iterative process of trial and error, the user can request that a loop invariant be tested by symbolic execution through the loop.

The interactive interface was designed and built at GRC Santa Barbara using a program which simulated the action of the Anagraph. The Anagraph simulation runs in batch mode on the 7600 and produces printed output which is the same as would be seen on the Anagraph screen. The use of the simulation greatly reduced the amount of time required to develop the interactive interface and allowed all but final testing to be done in batch mode from Santa Barbara.

When the verification condition generator and the simplifier of the Software Quality Laboratory are used interactively from the Anagraph terminal, commands are selected by the trackball and textual information is entered through the keyboard. Some commands may be entered by using the trackball alone while others (PATH, REPLACE, EXPRESSION, AND RXVP)[23] require the user to enter a text string through the keyboard. The interactive interface synthesizes the command corresponding to the command selected and places it in a command buffer which is displayed to the user when the ENTER button is depressed. The actual processing associated with the command takes place when the command GO is selected by the user. Textual output generated by a Software Quality Laboratory processing module is displayed on the Anagraph screen. As each page of output is displayed, the user can direct the interactive interface to display the next page of output, or to cease displaying the output and return to a mode where commands can be entered.

The figures in this section were derived from the Anagraph simulator.

The basic display is a menu of commands from which the user selects a command by placing the trackball position cursor over the command on the screen and pressing the trackball ENTER key. The command menu is shown in Fig. 6.8. Commands selected are echoed under the heading SELECTED COMMANDS. The seven commands that can be entered are described in the following paragraphs.

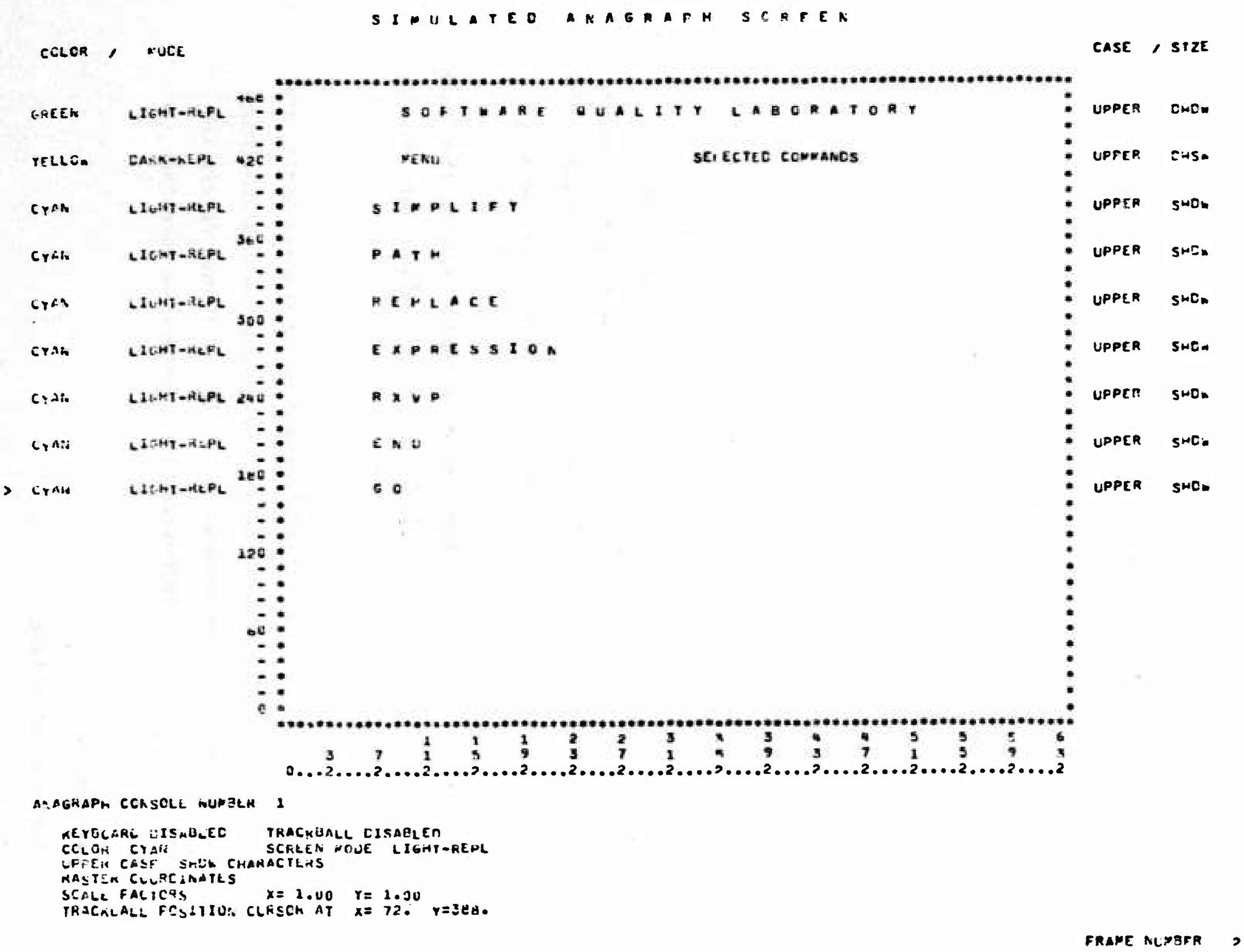
To select the SIMPLIFY command, the user places the trackball cursor over SIMPLIFY on the screen, and presses the trackball ENTER key. The command VCG,SIMPLIFY is constructed and echoed on the right half of the screen, under the heading SELECTED COMMANDS as shown in Fig. 6.9.

When the user selects the PATH command using the trackball, the interactive interface responds by printing the prompt ENTER PATH on the screen below the command menu. The user then enters the number of paths and path list (such as 2,1,2) through the Anagraph keyboard. At this point the screen appears as in Fig. 6.10. The command VCG,PATH = number of paths, path list is then constructed and entered in the command buffer by the interactive interface and echoed on the right half of the screen.

The PATH command causes the selection of a report showing the "path to verify," as shown in Fig. 6.11. This report replaces the menu display on the Anagraph screen.

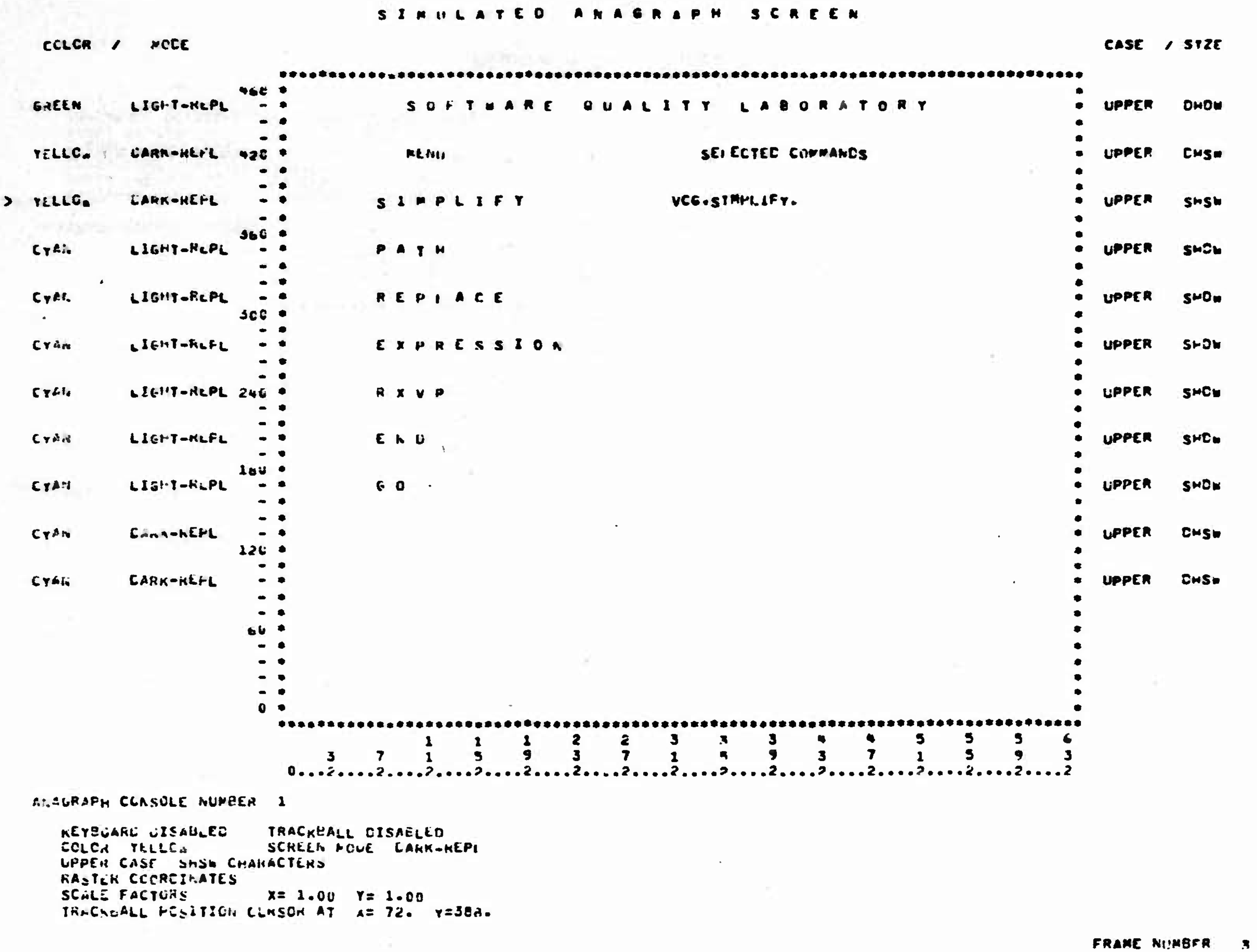
When the user selects the REPLACE command using the trackball, the interactive interface responds by printing the prompt ENTER REPLACEMENT STRING on the screen below the command menu. The user then enters the replacement string through the keyboard. The interactive interface then constructs the commands:

```
VCG,REPLACE
replacement string
*END.
```



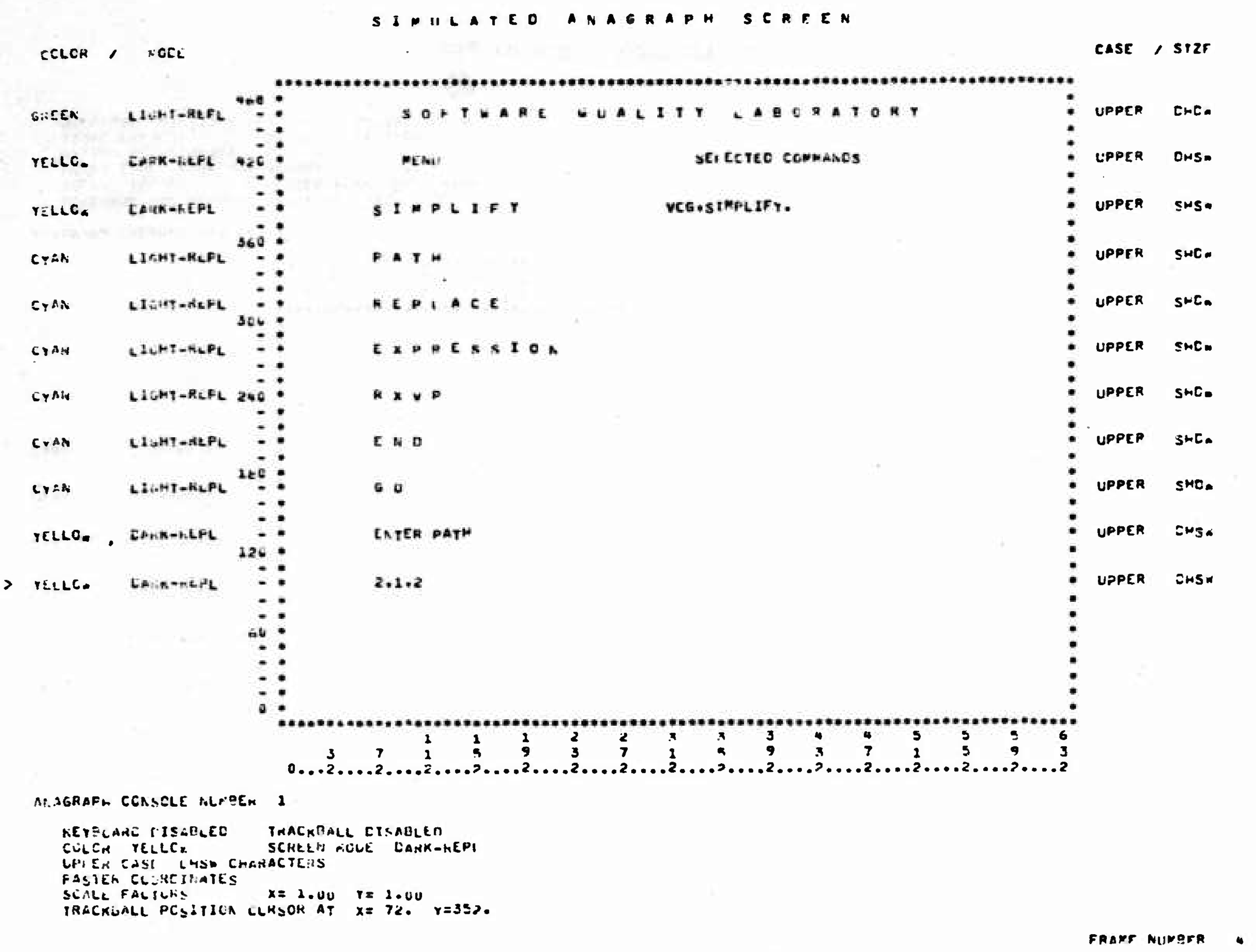
```
                      S I M U L A T E D   A N A G R A P H   S C R E E N

COLOR  /   MODE                                                                    CASE  / SIZE

                    480
GREEN     LIGHT-REPL  -       S O F T W A R E   Q U A L I T Y   L A B O R A T O R Y     UPPER   SHOW

YELLOW    DARK-REPL 420       MENU                    SELECTED COMMANDS                   UPPER   SHOW

CYAN      LIGHT-REPL  -     S I M P L I F Y                                               UPPER   SHOW
                    360
CYAN      LIGHT-REPL  -     P A T H                                                       UPPER   SHOW

CYAN      LIGHT-REPL  -     R E P L A C E                                                 UPPER   SHOW
                    300
CYAN      LIGHT-REPL  -     E X P R E S S I O N                                           UPPER   SHOW

CYAN      LIGHT-REPL 240    R X V P                                                       UPPER   SHOW

CYAN      LIGHT-REPL  -     E N D                                                         UPPER   SHOW
                    180
> CYAN    LIGHT-REPL  -     G O                                                           UPPER   SHOW

                    120

                     60

                      0
                               1   1   1   2   2   3   3   3   4   4   5   5   5   6
                       3   7   1   5   9   3   7   1   5   9   3   7   1   5   9   3
                   0...2...2...2...2...2...2...2...2...2...2...2...2...2...2...2...2

ANAGRAPH CONSOLE NUMBER  1

   KEYBOARD DISABLED     TRACKBALL DISABLED
   COLOR  CYAN           SCREEN MODE  LIGHT-REPL
   UPPER CASE  SHOW CHARACTERS
   MASTER COORDINATES
   SCALE FACTORS         X= 1.00  Y= 1.00
   TRACKBALL POSITION CURSOR AT  X= 72.  Y=388.

                                                                      FRAME NUMBER   2
```

Figure 6.8. Command Menu



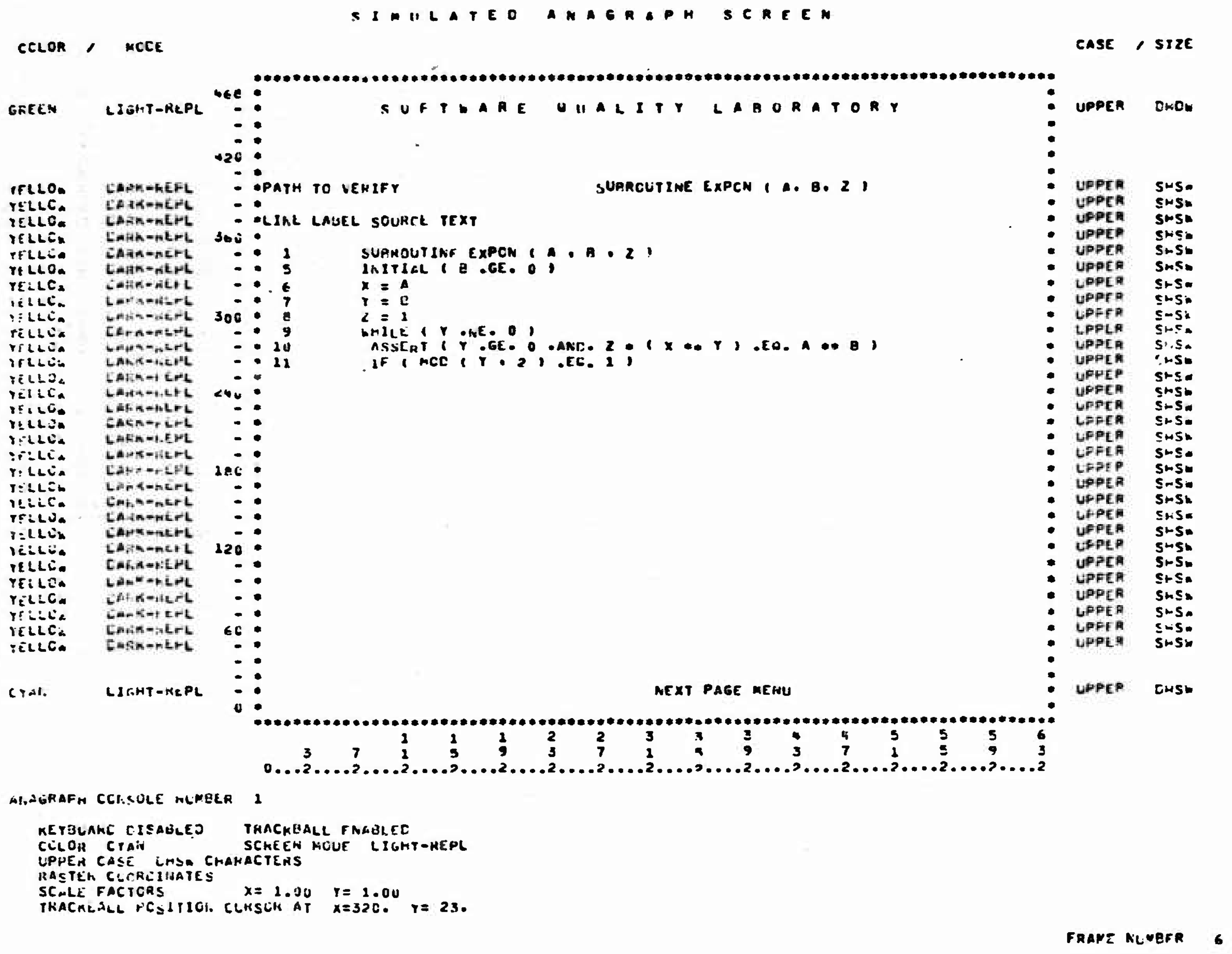
```
                              S I M U L A T E D   A N A G R A P H   S C R E E N
   COLOR  /   MODE                                                                              CASE  / SIZE
                         ******************************************************************
                     460 *                                                                *
   GREEN     LIGHT-REPL  - *     S O F T W A R E   Q U A L I T Y   L A B O R A T O R Y   *   UPPER   DHDW
                         - *                                                              *
                         - *                                                              *
   YELLOW    DARK-REPL   420 *     MENU                           SELECTED COMMANDS         *   UPPER   DHSW
                         - *                                                              *
                         - *                                                              *
 > YELLOW    DARK-REPL   - *   S I M P L I F Y             VCG.SIMPLIFY.                  *   UPPER   SHSW
                         - *                                                              *
                     360 *                                                                *
   CYAN      LIGHT-REPL  - *   P A T H                                                    *   UPPER   SHDW
                         - *                                                              *
                         - *                                                              *
   CYAN      LIGHT-REPL  - *   R E P I A C E                                              *   UPPER   SHDW
                     300 *                                                                *
                         - *                                                              *
   CYAN      LIGHT-REPL  - *   E X P R E S S I O N                                        *   UPPER   SHDW
                         - *                                                              *
                         - *                                                              *
   CYAN      LIGHT-REPL  240 *   R X V P                                                  *   UPPER   SHDW
                         - *                                                              *
                         - *                                                              *
   CYAN      LIGHT-REPL  - *   E N D                                                      *   UPPER   SHDW
                         - *                                                              *
                     180 *                                                                *
   CYAN      LIGHT-REPL  - *   G O                                                        *   UPPER   SHDW
                         - *                                                              *
                         - *                                                              *
   CYAN      DARK-REPL   - *                                                              *   UPPER   DHSW
                     120 *                                                                *
                         - *                                                              *
   CYAN      DARK-REPL   - *                                                              *   UPPER   DHSW
                         - *                                                              *
                         - *                                                              *
                         - *                                                              *
                      60 *                                                                *
                         - *                                                              *
                         - *                                                              *
                         - *                                                              *
                         - *                                                              *
                       0 *                                                                *
                         ******************************************************************
                                       1   1   1   2   2   3   3   3   4   4   5   5   5   6
                            3   7   1   5   9   3   7   1   5   9   3   7   1   5   9   3
                         0..2...2...2...2...2...2...2...2...2...2...2...2...2...2...2...2

   ANAGRAPH CONSOLE NUMBER  1

      KEYBOARD DISABLED     TRACKBALL DISABLED
      COLOR  YELLOW         SCREEN MODE  DARK-REPL
      UPPER CASE  SHSW CHARACTERS
      RASTER COORDINATES
      SCALE FACTORS         X= 1.00  Y= 1.00
      TRACKBALL POSITION CURSOR AT  X= 72.  Y=388.

                                                                                          FRAME NUMBER   3
```

Figure 6.9. SIMPLIFY Command



```
                         S I M U L A T E D   A N A G R A P H   S C R E E N

COLOR  /   MODE                                                                          CASE  / SIZE

                         ************************************************************
                     480 *
GREEN    LIGHT-REPL    - *    S O F T W A R E   Q U A L I T Y   L A B O R A T O R Y    *  UPPER   CHC.
                       - *                                                             *
                       - *                                                             *
YELLOW   DARK-REPL   420 *    MENU                          SELECTED COMMANDS          *  UPPER   DHS.
                       - *                                                             *
                       - *                                                             *
YELLOW   DARK-REPL     - *  S I M P L I F Y           VCG.SIMPLIFY.                    *  UPPER   SHS.
                       - *                                                             *
                     360 *                                                             *
CYAN     LIGHT-REPL    - *  P A T H                                                    *  UPPER   SHC.
                       - *                                                             *
                       - *                                                             *
CYAN     LIGHT-REPL    - *  R E P L A C E                                              *  UPPER   SHC.
                     300 *                                                             *
                       - *                                                             *
CYAN     LIGHT-REPL    - *  E X P R E S S I O N                                        *  UPPER   SHC.
                       - *                                                             *
                       - *                                                             *
CYAN     LIGHT-REPL  240 *  R X V P                                                    *  UPPER   SHC.
                       - *                                                             *
                       - *                                                             *
CYAN     LIGHT-REPL    - *  E N D                                                      *  UPPER   SHC.
                       - *                                                             *
                     180 *                                                             *
CYAN     LIGHT-REPL    - *  G O                                                        *  UPPER   SHD.
                       - *                                                             *
                       - *                                                             *
YELLOW , DARK-REPL     - *  ENTER PATH                                                 *  UPPER   CHS.
                     120 *                                                             *
                       - *                                                             *
> YELLOW DARK-REPL     - *  2.1.2                                                      *  UPPER   CHS.
                       - *                                                             *
                       - *                                                             *
                      60 *                                                             *
                       - *                                                             *
                       - *                                                             *
                       - *                                                             *
                       - *                                                             *
                       0 *                                                             *
                         ************************************************************
                                    1   1   1   2   2   3   3   3   4   4   5   5   5   6
                            3   7   1   5   9   3   7   1   5   9   3   7   1   5   9   3
                          0...2...2...2...2...2...2...2...2...2...2...2...2...2...2...2...2

ANAGRAPH CONSOLE NUMBER  1

   KEYBOARD DISABLED      TRACKBALL DISABLED
   COLOR  YELLOW          SCREEN MODE  DARK-REPL
   UPPER CASE  CHS. CHARACTERS
   RASTER COORDINATES
   SCALE FACTORS          X= 1.00  Y= 1.00
   TRACKBALL POSITION CURSOR AT  X= 72.  Y=352.

                                                                                  FRAME NUMBER   4
```

Figure 6.10. Enter PATH



```
                                S I M U L A T E D   A N A G R A P H   S C R E E N

 COLOR  /   MODE                                                                                              CASE  / SIZE

                          ************************************************************************************
                      480 *                                                                                  *
 GREEN      LIGHT-REPL  - *          S O F T W A R E   Q U A L I T Y   L A B O R A T O R Y                   *  UPPER   DHDW
                        - *                                                                                  *
                        - *                                                                                  *
                      420 *                                                                                  *
                        - *                                                                                  *
 YELLOW     DARK-REPL   - *PATH TO VERIFY                          SUBROUTINE EXPON ( A, B, Z )             *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *LINE LABEL SOURCE TEXT                                                            *  UPPER   SHSW
 YELLOW     DARK-REPL 360 *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *  1        SUBROUTINE EXPON ( A , B , Z )                                         *  UPPER   SHSW
 YELLOW     DARK-REPL   - *  5        INITIAL ( B .GE. 0 )                                                   *  UPPER   SHSW
 YELLOW     DARK-REPL   - *  6        X = A                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *  7        Y = B                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL 300 *  8        Z = 1                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *  9        WHILE ( Y .NE. 0 )                                                     *  UPPER   SHSW
 YELLOW     DARK-REPL   - * 10         ASSERT ( Y .GE. 0 .AND. Z * ( X ** Y ) .EQ. A ** B )                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - * 11         IF ( MOD ( Y , 2 ) .EQ. 1 )                                           *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL 240 *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL 180 *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL 120 *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL  60 *                                                                                  *  UPPER   SHSW
 YELLOW     DARK-REPL   - *                                                                                  *  UPPER   SHSW
                        - *                                                                                  *
                        - *                                                                                  *
> CYAN      LIGHT-REPL  - *                                        NEXT PAGE MENU                            *  UPPER   DHSW
                        0 *                                                                                  *
                          ************************************************************************************
                                         1    1    1    2    2    3    3    3    4    4    5    5    5    6
                               3    7    1    5    9    3    7    1    5    9    3    7    1    5    9    3
                          0...2....2....2....2....2....2....2....2....2....2....2....2....2....2....2....2

 ANAGRAPH CONSOLE NUMBER  1

    KEYBOARD DISABLED     TRACKBALL ENABLED
    COLOR  CYAN           SCREEN MODE  LIGHT-REPL
    UPPER CASE  DHSW CHARACTERS
    RASTER COORDINATES
    SCALE FACTORS          X= 1.00  Y= 1.00
    TRACKBALL POSITION CURSOR AT  X=320.  Y= 23.

                                                                                                      FRAME NUMBER   6
```

Figure 6.11. Report from Path Command

enters them in the command buffer, and echoes them on the right half of the screen.

The REPLACE command causes the generation of a report showing the result of applying the simplification rule to the verification condition being simplified. This report replaces the menu display on the Anagraph screen.

When the user selects the EXPRESSION command, using the trackball, the interactive interface responds by printing the prompt ENTER EXPRESSION on the screen below the command menu. The user then enters the expression through the keyboard. The interactive interface then constructs the commands

```
VCG,EXPRESSION
expression
*END.
```

enters them in the command buffer, and echoes them on the right half of the screen.

The RXVP command is used to enter commands to the Software Quality Laboratory which do not appear on the command menu. When the user selects it, using the trackball, the interactive interface responds by printing the prompt ENTER COMMAND on the screen below the command menu. The user then enters the command through the keyboard. The interactive interface enters the command into the command buffer and echoes it on the right half of the screen. Any valid Software Quality Laboratory command may be entered in this fashion.

The reports **corresponding** to the entered command are produced on the Anagraph screen.

The END command is used to close all files, print a final report, and terminate the execution of the Software Quality Laboratory. When the user selects it, using the trackball, the interactive interface places the command in the command buffer and echoes it on the right half of the screen.

The END command causes the generation of a wrapup report, which lists the modules on the library and their attributes, along with statistics on library creation and access.

The GO command causes the interactive interface to transfer control to the Software Quality Laboratory's command-processing module. All commands that have been previously selected (and echoed on the screen under SELECTED COMMANDS) will then be executed, and the first page of the resulting reports will be displayed on the Anagraph screen.

As each screenful of Quality Laboratory output is displayed on the Anagraph, the user is given the option of viewing the next page of output or returning to the command menu. These options are presented to the user by the words NEXT PAGE and MENU at the bottom of the screen. To select the next page of output or return to the command menu, the user places the trackball cursor over the appropriate command and presses the trackball ENTER key. If NEXT PAGE is selected and there is no more output, the command menu is displayed.

6.3 VERIFIED PROGRAMS

The formal verification process has evolved from the generation of verification conditions which required manual simplification to a process which requires human interaction.

Included in Appendix D are listings of programs for which verification conditions have been generated. Also included in a few cases are the simplified verification conditions and the reduction via the user-supplied simplification.

These initial programs were chosen so that comparisons could be made between the verification conditions that were generated by the Software Quality Laboratory and similar verification conditions generated by others.[5-9] It is recognized that others have attempted to have their conditions automatically reduced to .TRUE., whereas our approach attempts to present them to a user in readable form so that the user can modify them. This may be why the output shown in the appendix is more readable.

The first nine programs are from King.[5] They were also used by Deutsch[6] and a few by Elspas.[7] The tenth program has been used by Elspas.[7] The eleventh program has been used by several others.[6,9] The twelfth program is new.

1. Program TIMES computes an output X = A * B by adding the input A to a local variable SUM, B times. The local variable Y is used as a counter which is initially set to B and then decremented to zero.

2. Program DIV computes two outputs: Q, which is the quotient of the integer division of A by B, and R, which is the remainder of that division. A and B are inputs.

3. Program EXPON computes an output Z = A ** B by multiplying the partial result times itself using the binary value of the input B as found by the MOD function. The local variable X is used as the partial result and Y is used to find the binary representation for B.

4. Program PRIME computes whether the input variable A is a prime number and sets the output J to a "0" if A is prime of a "1" if A is not prime. Testing is done by taking the remainder of A divided by 2 to A - 1 using the local counter I.

5. Program ZERO sets an array A of length N to zero. A is treated as input and output. N is input. The local variable I is used as a counter.

6. Program MAXI searches an array A of length N for the largest element. This element with the largest value in the array is then swapped with the element A(N). A is treated as input and output. N is input. The local variable I is used as a counter and the local variable TEMP is used as temporary storage during the swap.

7. Program SORT1 performs a sort of an array A of length N. Elements are exchanged using the local variable TEMP. I is a local variable used as a counter. J is a local variable which when set to 1 indicates a swap was made. J is 0 when sorting is no longer necessary. A is used as input and output. N is used as input.

8. Program MULT2 is a more complex version of the TIMES program which performs multiplication of A times B whether A or B is negative. The result is placed in ANS which is an output variable. A and B are input variables. TEMPA, TEMPB and TANS are used as temporary variables.

9. Program SORT2 also performs a sort of the array A of length N. The sort is accomplished by finding the largest element in the rest of array at each iteration. Local variables I, J and K are used as counters. Local variables M, N and L are used as assertion counters. Local variable TEMP is used for swapping. A is used as input and output. N is used as input.

10. Program BINSCH performs a binary search of an array ARRAY of length LENGTH for the value in X. The element index where X is located is placed in the output variable LOOKUP. If not found the output variable ERROR is set to .TRUE., otherwise it is set to .FALSE.. ARRAY, X, and LENGTH are treated as input variables. I is a local variable used as a counter. M and N are local variables used to delimit the area to be searched each time. SORTED is a function of an assertion on the array, used to provide a more readable version of the assumption that the array is SORTED on input.

11. Program FIND is an efficient sorting algorithm published by Hoare.[24] In this version A is the array of length NN which is sorted about element A(F). A is used as input and output. NN and F are used as input. I, J, M, N are used as counters in the algorithm. P and Q are used as counters in the assertions. PSORT is an assertion function used to provide a more readable version of the assertion that the array is partially sorted.

12. Program SQX is a square root algorithm using Newton's method to find an approximation to the square root of the floating point input variable X . The program is a function where Y is used as a local variable to represent the approximation.

7 SOFTWARE DEVELOPMENT SYSTEM QUALITY ASSESSMENT

In this section we shall examine the role of high-level languages in the production of reliable BMD software. Our approach is to use a language developed for multi-tasking software systems, Concurrent PASCAL,[25] in reprogramming an existing BMD software simulation. The BMD simulation, GSIM, has been described in previous reports.[16] As each algorithm from GSIM is implemented in Concurrent PASCAL, assertions will be derived which express the correct behavior of the algorithm. From these assertions we will derive verification conditions, simplify them, and prove them correct.

7.1 THE APPLICABILITY OF CONCURRENT PASCAL TO BMD SOFTWARE

We have found GSIM algorithms involving concurrent operations to be easily constructed in Concurrent PASCAL. However, we have also discovered certain task sequencing requirements common to BMD software which are not expressible in Concurrent PASCAL. Before we discuss these, we will briefly describe the Concurrent PASCAL Monitor and its use in implementing an example algorithm from GSIM.

The language structure for expressing concurrent operations in Concurrent PASCAL is the monitor.[26] A monitor is a single programming unit consisting of a shared data structure, local variables, procedures and/or functions which operate on the shared data structure, Delay and Continue operations and initialization statements. The monitor which controls access to the search return data set of GSIM is shown in Fig. 7.1. The monitor schedules exclusive access to the shared data structure when a call is made upon one of the monitor procedures by a concurrently executing process (task). This scheduling is done by the virtual machine or operating system routine which implements the monitor. Access to the monitor is granted on a first come-first served basis. The procedures which are defined within the monitor are local to the monitor and have access only to data which is local to the monitor. Variables which are local to the monitor can only be manipulated by calling the

```
SRDSMON = MONITOR ( LIMIT : INTEGER ) ;

VAR  RTNQ = ARRAY[0..LIMIT-1] OF SEARCHRETURN ;
     SENDER, RECEIVER : QUEUE ;
     HEAD, TAIL, LENGTH : INTEGER ;

PROCEDURE ENTRY PUT ( RETURN : SEARCHRETURN ) ;
BEGIN
   IF LENGTH = LIMIT THEN DELAY (SENDER) ;
   RTNQ[TAIL] := RETURN ;
   TAIL := (TAIL + 1) MOD LIMIT ;
   LENGTH := LENGTH + 1 ;
   CONTINUE (RECEIVER) ;
END (* PUT *) ;

PROCEDURE ENTRY GET (VAR RETURN : SEARCHRETURN) ;
BEGIN
   IF LENGTH = 0 THEN DELAY (RECEIVER) ;
   RETURN := RTNQ[HEAD] ;
   HEAD := (HEAD + 1) MOD LIMIT ;
   LENGTH := LENGTH - 1 ;
   CONTINUE (SENDER) ;
END (* GET *) ;

BEGIN (* INITIALIZE *)
   HEAD := 0 ;
   TAIL := 0 ;
   LENGTH := 0
END (* SRDS MONITOR *) ;
```

Figure 7.1. Monitor for Search Return Data Set

monitor procedures. When a process or task which has been given access to the monitor executes a Delay operation, the task is suspended and exclusive access to the monitor can be given to another task. Tasks which have been suspended are reactivated on a first come-first served basis when a Continue operation is performed. In summary, monitors implement mutual exclusion of concurrent tasks when they operate on a shared data structure by allowing only sequential access to the data structure. They prevent tasks from performing incorrect operations on the shared data structure by requiring explicit definitions of the allowed operations, and synchronize cooperating tasks through Delay and Continue operations.

We will now give an example of a monitor which controls the access to the data structure shared by concurrent tasks. Figure 7.2 shows how a verify pulse is generated in response to a search return in GSIM. Radar returns are placed in the radar returns buffer (RRB) by the channel controller task. The assimilate radar returns task then classifies the returns according to type and places all search returns in the search returns data set (SRDS). The generate verify pulse task generates radar requests for verify pulses from the search returns and places these requests in the radar activities queue (RAQ). The generate radar orders task then generates radar orders from the radar requests. Notice that the SRDS is accessed by both the assimilate radar returns task and the generate verify pulse. This is the data set we will implement with a monitor.

Figures 7.1, 7.3 and 7.4 show the Concurrent PASCAL implementation of the search return data set monitor, the assimilate radar returns process, and the generate verify pulse process, respectively. The assimilate radar returns process is a cyclic task which gets radar returns from the RRB, classifies them as to type and in the case of search returns, places them in the SRDS using the monitor call SRDS.PUT. The generate verify returns task is also a cyclic task which retrieves search returns from the SRDS using the monitor call SRDS.GET and creates radar requests for the radar activities queue. The SRDS monitor implements a queue of radar returns using an array data structure. The monitor contains a procedure to place entries in the queue and another procedure to retrieve entries from the queue.

It is possible for the assimilate radar returns process and the generate verify pulse process to attempt to access the SRDS simultaneously since they are independent tasks and we have made no assumption about their relative speeds. The monitor prevents this from occurring by allowing only one of the processes to complete a call to a monitor procedure at one time. Recall that this mutual exclusion property is

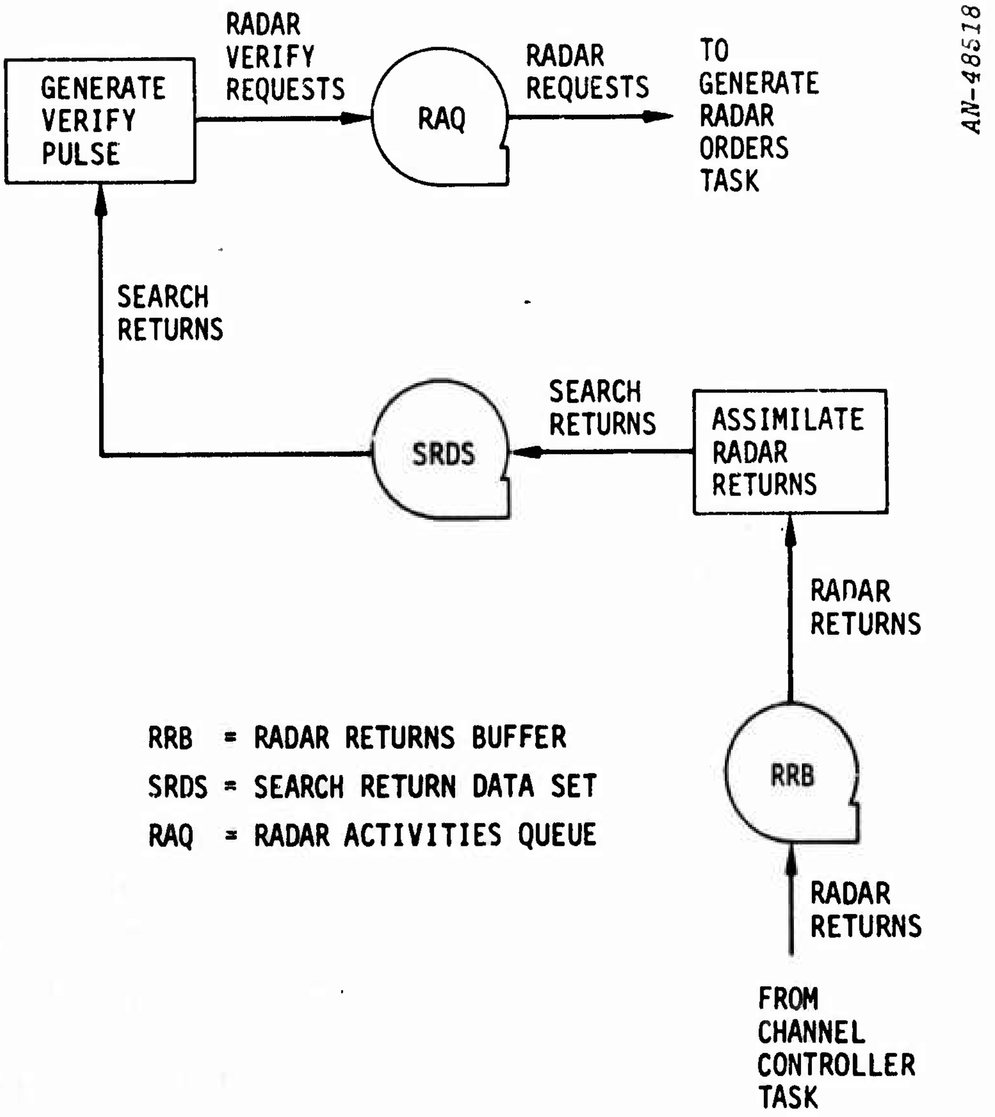


Figure 7.2. Search Return Processing in GSIM

```
ASSIM = PROCESS
      (TOTDS : TOTDSMON ; RRB : RRBMON ; OTDS : OTDSMON ;
       SRDS : SRDSMON ; TRACKFILE : RESOURCE ) ;

VAR  OTDSREC, TOTDSREC, LOTDS : TRACKRECORD ;
      SRDSREC : SEARCHRETURN ;
      RETURN : RADARRETURN ;

PROCEDURE MAKEOTDSREC ;
BEGIN
   OTDS.OTID := OBJID ;
   OTDS.OTTIME := RETURN.MEASTIME - RTTRK ;
   OTDS.OTRNGE := RETURN.RANGE - RTTRK*RCLNGM
END (* MAKE OTDS RECORD *) ;

PROCEDURE MAKETOTDSREC ;
BEGIN
   TOTDS.CTID := RETURN.ID ;
   TOTDS.CTBEAM := RETURN.BEAMPOS ;
   TOTDS.CTPUL := RETURN.PULSETYPE ;
   TOTDS.CTTIME := RETURN.MEASTIME ;
   TOTDS.CTRNGE := RETURN.RANGE ;
   TOTDS.CTQUAL := RETURN.RTNQUAL
END  (* MAKE TOTDS RECORD *) ;

BEGIN
CYCLE
   RRB.GET ( RETURN ) ;
   WITH RETURN DO BEGIN
      CASE PULSETYPE OF
         SEARCH :
            BEGIN
               SPOUT := SPOUT + 1 ;
               IF RTNQUAL = GOOD THEN
               BEGIN (* CREATE AN ENTRY IN THE SRDS *)
                  SRDSREC.BEAMPOS := BEAMPOS ;
                  SRDSREC.RANGE := RANGE ;
                  SRDSREC.MEASTIME := MEASTIME ;
                  SRDS.PUT ( SRDSREC )
               END (* IF *)
            END (* SEARCH PULSE *) ;
         VERIFY :
            BEGIN (* CREATE ENTRIES IN OTDS AND TOTDS *)
               IF RTNQUAL = GOOD THEN
               BEGIN
                  OBJID := OBJID + 1 ;
                  ID := OBJID ;
                  MAKEOTDSREC ;
                  MAKETOTDSREC ;
                  TOTDS.PUT ( TOTDSREC ) ;
                  TRACKFILE.REQUEST;
                  OTDS.PUT ( OTDSREC ) ;  *
                  TRACKFILE.RELEASE
               END (* IF *)
            END (* VERIFY PULSE *) ;
         SPECIALSEARCH :
            BEGIN  (* CREATE ENTRY IN TOTDS OR LOTDS *)
               IF RTNQUAL = GOOD THEN
               BEGIN
                  MAKETOTDSREC ;
                  TOTDS.PUT ( TOTDSREC )
               END
               ELSE  (* TRACK LOST *)
               BEGIN
                  TRACKFILE.REQUEST;
                  OTDS.MATCH ( ID : OTDSREC ) ;
                  TRACKFILE.RELEASE ;
                  LOTDSREC := OTDSREC ;
               END (* IF *)
            END (* SPECIAL SEARCH PULSE *) ;
         TRACK :
            BEGIN  (* CREATE ENTRIES IN TOTDS *)
                  MAKETOTDSREC ;
                  TOTDS.PUT ( TOTDSREC ) ;
            END (* TRACK PULSE *)
      END (* CASE *)
   END (* WITH *)
END (* CYCLE *)
END (* ASSIMILATE RADAR RETURNS *) ;
```

Figure 7.3. Assimilate Radar Returns Process

```
VERGEN = PROCESS (RAQ : RADMON ; SRDS : SRDSMON ) ;

VAR  REQUEST : RADARACTIVITY ;
     RETURN : SEARCHRETURN ;

BEGIN
CYCLE
   SRDS.GET (RETURN) ;
   WITH REQUEST, RETURN DO
   BEGIN
      ID := NXTSP ;
      NXTSP := NXTSP + 1 ;
      PULSETYPE := VERIFY ;
      REQUEST.BEAMPOS := RETURN.BEAMPOS ;
      XMITTIME := TIME + PINOM ;
      RECVTIME :=    PDURAT(VFY)
                 + 2*(RANGE + RCLNUM * (XMITTIME-MEASTIME)) / VLITE
                 - ( RANDOM(VFY) / 2 )
   END (* WITH) ;
   RAQ.PUT (REQUEST)
END (* CYCLE*)
END (* GENERATE VERIFY PULSE *) ;
```

Figure 7.4. Generate Verify Pulse Process

implicit in the definition of the monitor. The monitor also prevents either process from attempting to place or retrieve data in storage locations beyond the length of the queue. For example, in the PUT procedure, if the queue is full (LENGTH = LIMIT) then the assimilate radar returns process is delayed until the generate verify pulse process retrieves a search return from the queue. Similarly, in the GET procedure, the generate verify pulse process is delayed when the queue is empty (LENGTH = 0) and is only allowed to continue after the assimilate radar returns process has placed a radar return in the queue. Therefore, we see that the monitor has been successful in solving the problem of exclusive access to the search return data set and at the same time has synchronized the interaction of the assimilate radar returns process and the generate verify pulse process.

We shall now use GSIM to illustrate a number of problems that cannot be solved using the monitor construct. Figure 7.5 depicts the relationships between the tasks and data structures used in the generation and processing of radar requests. The generate search pulse, generate verify pulse and generate track pulse tasks all place requests for the radar in the radar activities queue (RAQ). The generate radar orders task takes these requests from the RAQ and generates a radar schedule satisfying these requests which does not exceed the constraints of the radar. Unscheduled requests are returned to the RAQ. As will be explained in the next section, this relationship is an example of the reader's/writer's problem in concurrent processing. A read operation in this case removes a request from the queue. There are more writers than readers; however, in a BMD software system it would be preferable to let the reader have priority over any of the writers. Suppose we implement the RAQ with a monitor and consider the case where two of the three writing processes are awaiting access to the RAQ while the third writer is using it. If the generate radar orders task makes a read request, it will have to wait until both write requests have been processed. The manner in which exclusive access is granted by the monitor (first come-

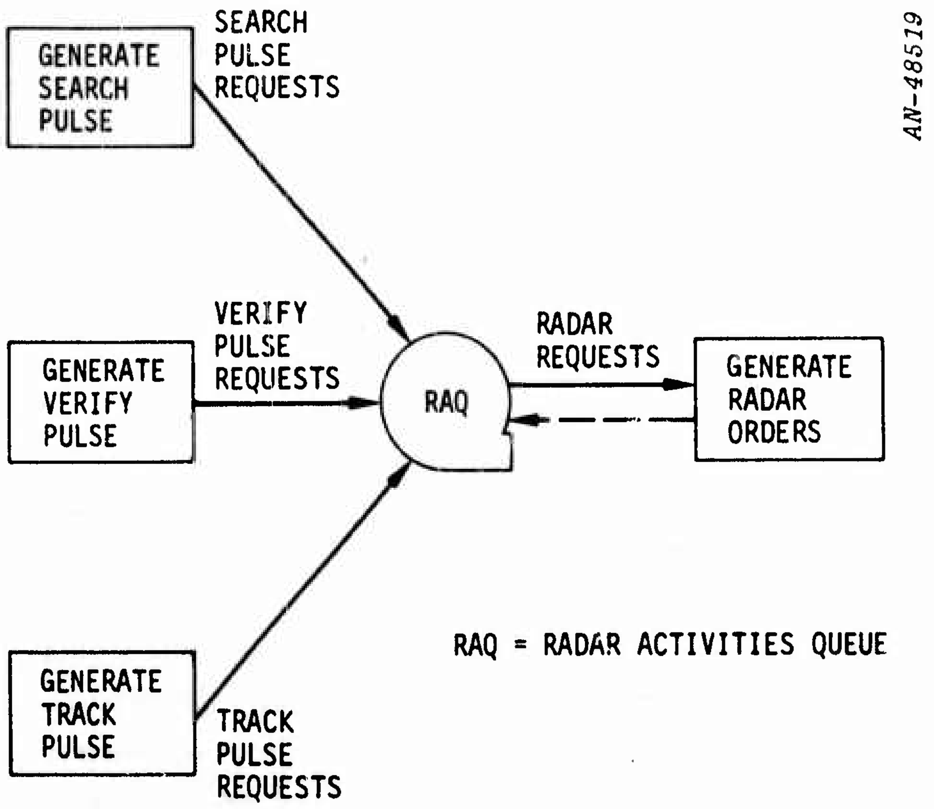
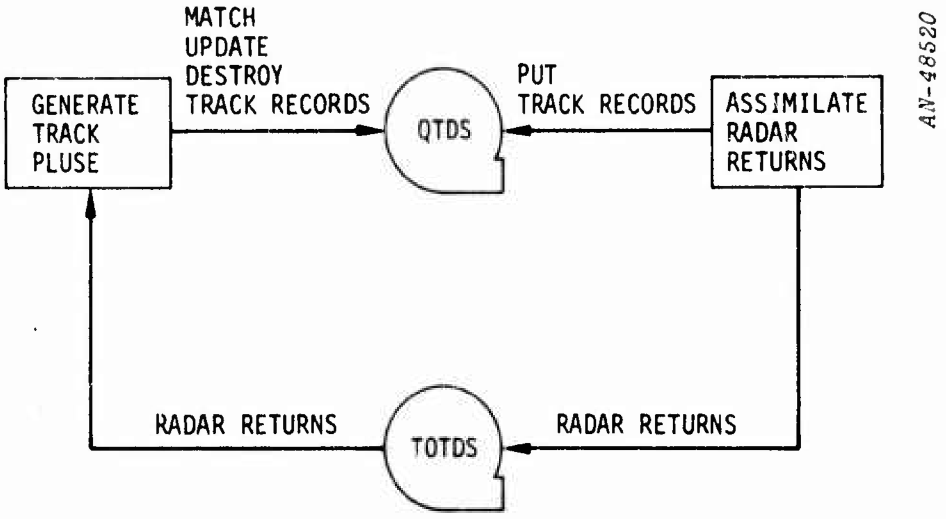


Figure 7.5. Radar Activity Processing in GSIM

first served) prevents the generate radar orders task from getting access to the RAQ immediately after the writer which is using the queue has finished. However, since radar time is a scarce resource, we may not want the generate radar orders task to wait for those tasks which are generating radar requests. Unfortunately, there is no facility in the Concurrent PASCAL monitor for implementing this priority arrangement.

As a second example, consider the fact that we allowed the generate radar orders task to get radar requests from and put radar requests into the RAQ. This is perfectly reasonable in this case. Now consider the search return processing which was depicted in Fig. 7.2. It would not be reasonable in this case to allow the assimilate radar returns process to both put and get search returns from the SRDS. There is nothing in the description of a monitor which protects us from this programming error. That is, the monitor construct has no mechanism to protect the shared data structure from being accessed by a process using an improper operation.

Figure 7.6, which depicts track processing in GSIM illustrates another problem with the monitor construct. The object track data set (OTDS) is the track file for GSIM. The OTDS is shared by the assimilate radar returns process which places track records in the OTDS and the generate track pulse process which processes these track returns and generates track pulses for the radar. These two processes also share the temporary object track data set which is a queue of radar returns from which the generate track pulse process matches entries in the OTDS. The generate track pulse process is allowed to perform three operations on the track file: match a track record with a radar return, update a track record, and destroy a track record. However, the monitor implementation of the OTDS does not allow us to specify the order in which these operations are to take place. If, due to a programming error in the generate track pulse process, a destroy operation were to precede a



OTDS = OBJECT TRACK DATA SET OR TRACK FILE
TOTDS = TEMPORARY OBJECT TRACK DATA SET

Figure 7.6. Track Processing in GSIM

match operation, the wrong track record could be destroyed. Although there are ways in which the monitor procedures may be programmed to check for an invalid order of operations, there is no way to state these constraints explicitly in the monitor description.

7.2 CLASSES OF CONCURRENCY

In this section we shall identify several classes of concurrency problems, relate them to problems that have been studied previously and identify the types of concurrency which occur in GSIM.

By studying the types of concurrent processing problems discussed in the computer science literature.[27-32] we can abstract at least four dimensions along which to classify these problems. Figure 7.7 shows these dimensions. From the first two dimensions, the number of processes which are allowed to access the data structure simultaneously and the number of shared data structures which each process requires, we can

THE NUMBER OF PROCESSES GRANTED SIMULTANEOUS ACCESS

ONE

MORE THAN ONE

THE NUMBER OF SHARED DATA STRUCTURES REQUIRED SIMULTANEOUSLY BY A PROCESS

ONE

MORE THAN ONE

THE TYPES OF ACCESS ALLOWED TO THE SHARED DATA STRUCTURE

WRITE, PRODUCE OR CREATE

READ OR USE

UPDATE OR CHANGE

DESTROY OR CONSUME

THE SCHEDULING RULE BY WHICH THE SHARED DATA STRUCTURE IS ALLOCATED

FIRST COME-FIRST SERVED

PRIORITY

OTHER

Figure 7.7. Dimensions of Concurrent Processing Problems

identify four problems in concurren* processing. These are summarized in Fig. 7.8. In the case where only one user is allowed exclusive access to a single shared data structure, we have the mutual exclusion problem. This problem is typically present in tasks with producer/consumer relationships. If more than one process can be given access to a shared data structure at a time, we have the problem of mutual exclusion between classes of processes. Problems of this type usually occur when processes can be classified into those which read from a data base and those which write into a data base. If a single process requires more than one shared data structure at a time, then we have a cooperation problem. These are resource allocation problems in which the major error to be avoided is deadlock. Finally if more than one process can access a shared data structure simultaneously and each process must access more than one shared data structure, we have a problem in class cooperation.

		NUMBER OF SIMULTANEOUS USERS	
		1	>1
NUMBER OF DATA STRUCTURES REQUIRED	1	MUTUAL EXCLUSION	MUTUAL EXCLUSION OF CLASSES
	>1	COOPERATION	CLASS COOPERATION

AN-48521

Figure 7.8. Problems in Concurrent Processing

The operation a process performs on a shared data structure is another dimension by which concurrent programming problems can be classified. In the most general sense, a process can create (write) an element in a data structure; read an element of a data structure; change an element of a data structure; or destroy (consume) an element of a data structure.

Lastly, the scheduling rule which we use to allocate exclusive access to the data structure can be used to differentiate between types of concurrent programming problems. In the case of the monitor, the processes which request access to the data structure are given access to it on a first come-first served basis. However, as we have shown previously, other scheduling rules may be desired.

From this simple classification scheme for concurrent programming problems, we can identify which problems occur in GSIM. Figure 7.9 shows the GSIM tactical software processes and the data sets that they access, and Fig. 7.10 classifies the ways in which each data structure is used. The SRDS and TOTDS are producer/consumer problems and were easily implemented using the monitor construct. The RAQ and OTDS however were data sets for which we identified difficulties in the monitor implementation. These data sets exhibit requirements for priority access and multiple operations, respectively. There are a number of examples of concurrent processing which do not occur in GSIM, however, we expect that all of the types of concurrency will be present in a BMD software system. For example, if there were multiple processes which assimilated radar returns and a number of processes which generated verify pulses, we would have classified the SRDS as a problem of mutual exclusion of classes with no priority.

7.3 ASSERTIONS FOR CONCURRENT PASCAL MONITORS

In this section we develop assertions for the search return data set monitor in GSIM. But first we identify several general requirements

AN-48522

		EXCLUSION CLASS	OPERATIONS	SCHEDULING RULES
GSIM DATA SET	RAQ	MUTUAL EXCLUSION OF CLASSES	PRODUCE, CONSUMES, PRODUCE	READ PRIORITY
	SRDS	MUTUAL EXCLUSION	PRODUCE, CONSUME	NO PRIORITY
	TOTDS	MUTUAL EXCLUSION	PRODUCE, CONSUME	NO PRIORITY
	OTDS	MUTUAL EXCLUSION	CREATE, READ-UPDATE-DESTROY	NO PRIORITY

Figure 7.9. Classification of Concurrency in GSIM

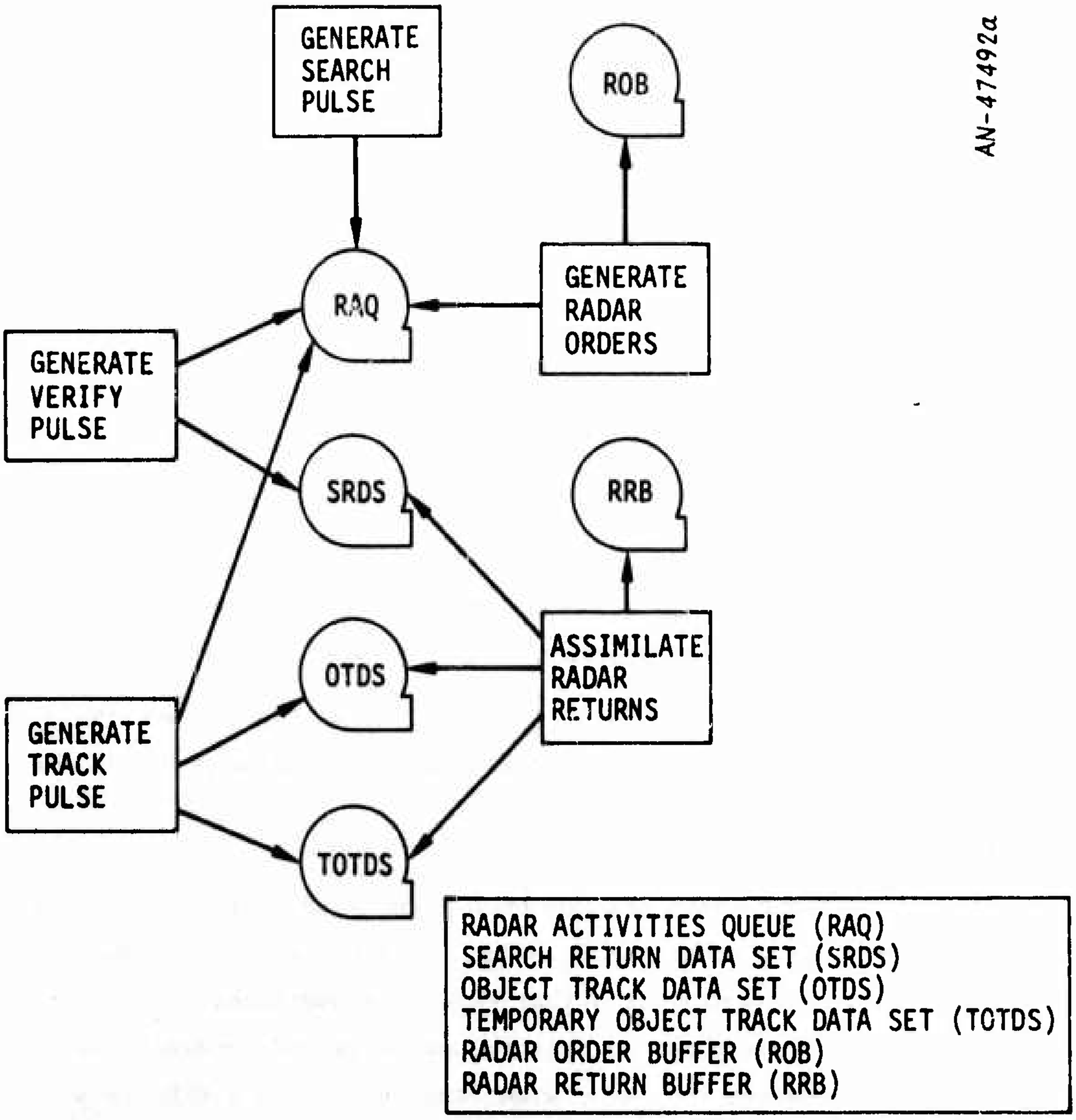


Figure 7.10. GSIM Process and Data Structures

for the correctness of any concurrent processing implementation. From these we define correctness criteria for the class of problem the SRDS illustrates. Finally we describe a symbolic execution technique for deriving verification conditions from these assertions.

In Ref. 27, Dijkstra identifies four criteria for the correctness of a solution to a mutual exclusion problem. They are:

- Mutual exclusion - only one process at a time may have access to a shared data structure
- No deadlock - when a shared resource is requested simultaneously by several processes, it must be granted to one of them within a finite time
- No starvation - when a process acquires a shared resource, the process must release it again within a finite time
- No busy waiting - a process should not consume processing time while it is waiting to acquire a shared resource

In addition, our solution should not make any assumptions about the relative speed of the processes since in general we cannot predict the order in which processes will be executed.

The search return data set has been identified as the shared data structure of a producer/consumer problem. To prove the correctness of our monitor solution we must show that several assumptions are not violated. The first of these is that the producer and consumer are prevented from accessing the queue simultaneously. Since this is an implicit assumption when the queue is implemented using a monitor, we cannot prove this assumption without appealing to how the monitor construct is implemented. Secondly, we must show that the data structure behaves as a queue. That is, messages cannot be taken from the queue when it is empty, a new message cannot be placed in the queue when it is full and that any message placed in the queue will eventually be retrieved from the queue. Thirdly, we must prove the "no blocking" criterion: both the producer and consumer cannot be waiting for each other simultaneously. For the monitor implementation, this implies that for every DELAY statement in a monitor procedure there must exist a corresponding CONTINUE statement and that if any DELAY statement has been executed, then the path which contains the corresponding CONTINUE statement can be executed to remove the DELAY. Finally, we must show that neither the producer nor consumer can continually overtake the other by always gaining access to the shared data structure, that is, that the scheduling of the monitor is fair.

We will now derive the assertions which describe the correct behavior of the SRDS monitor. The first assertion is the invariant for the correct operation of the queue. As shown in Fig. 7.1, the queue is implemented by an array of search returns whose maximum index is LIMIT - 1. The element at the front of the queue is indicated by the index HEAD and the element at the end of the queue is indicated by the index TAIL. Since elements are inserted into the array in a circular manner (after an element has been entered in position LIMIT - 1 the next element will be entered in position 0), the length of the queue is given by the expression:

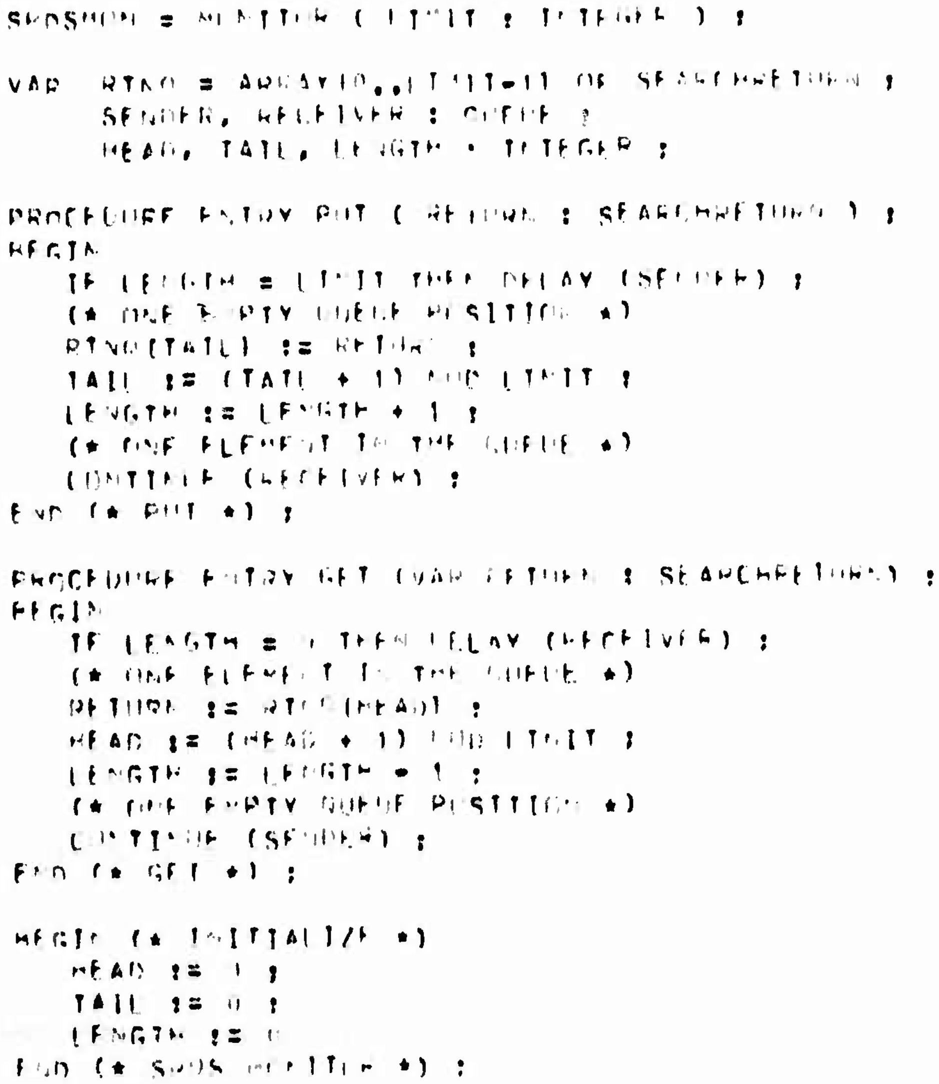
$$\text{LENGTH} = \text{ABS}(\text{TAIL} - \text{HEAD}) + 1$$

Therefore, the invariant assertion for correct operation of the queue is

$$0 \leq \text{LENGTH} \leq \text{LIMIT AND LENGTH} = \text{ABS}(\text{TAIL} - \text{HEAD}) + 1$$

This assertion must always be true in the monitor.

We can however make stronger statements concerning the number of elements in the queue after DELAY statements and before CONTINUE statements. Figure 7.11 shows the SRDS monitor with comments which indicate the conditions which must be true after DELAY and before the CONTINUE statements in the PUT and GET procedures. In the PUT procedure, the



```
SRDSMON = MONITOR ( LIMIT : INTEGER ) ;

VAR  RING = ARRAY[0..LIMIT-1] OF SEARCHRETURN ;
     SENDER, RECEIVER : QUEUE ;
     HEAD, TAIL, LENGTH : INTEGER ;

PROCEDURE ENTRY PUT ( RETURN : SEARCHRETURN ) ;
BEGIN
   IF LENGTH = LIMIT THEN DELAY (SENDER) ;
   (* ONE EMPTY QUEUE POSITION *)
   RING[TAIL] := RETURN ;
   TAIL := (TAIL + 1) MOD LIMIT ;
   LENGTH := LENGTH + 1 ;
   (* ONE ELEMENT IN THE QUEUE *)
   CONTINUE (RECEIVER) ;
END (* PUT *) ;

PROCEDURE ENTRY GET (VAR RETURN : SEARCHRETURN) ;
BEGIN
   IF LENGTH = 0 THEN DELAY (RECEIVER) ;
   (* ONE ELEMENT IN THE QUEUE *)
   RETURN := RING[HEAD] ;
   HEAD := (HEAD + 1) MOD LIMIT ;
   LENGTH := LENGTH - 1 ;
   (* ONE EMPTY QUEUE POSITION *)
   CONTINUE (SENDER) ;
END (* GET *) ;

BEGIN (* INITIALIZE *)
   HEAD := 1 ;
   TAIL := 0 ;
   LENGTH := 0
END (* SRDS MONITOR *) ;
```

Figure 7.11. Search Return Data Set Monitor with Comments

sending process is delayed if the queue is full. This process cannot be resumed until the GET procedure is called and the CONTINUE operation is executed indicating at least one empty position in the queue. Conversely, the process which calls the GET procedure is delayed if the queue is empty. It cannot be restarted until the PUT procedure is called and its CONTINUE operation is executed signaling at least one element in the queue. The assertions which must be placed after the DELAY statements and before the CONTINUE statements in order to express these relationships are shown in Fig. 7.12. In general, the assertions which must surround a DELAY statement can be expressed as

$$\{I \wedge \neg B\} \text{ DELAY } \{I \wedge B\}$$

and those which surround a CONTINUE statement as

$$\{I \wedge B\} \text{ CONTINUE } \{I\}$$

In these assertions, B expresses the condition which must be satisfied before a task can be reactivated following a DELAY operation. The assertion I (the monitor invariant) must be true whenever the locus of control changes in the monitor. Since any monitor procedure can be executed after a DELAY operation, this invariant must be true on entry to any monitor procedure. In the case of the SRDS monitor, the invariant is simply our first expression describing the correct operation of the queue.

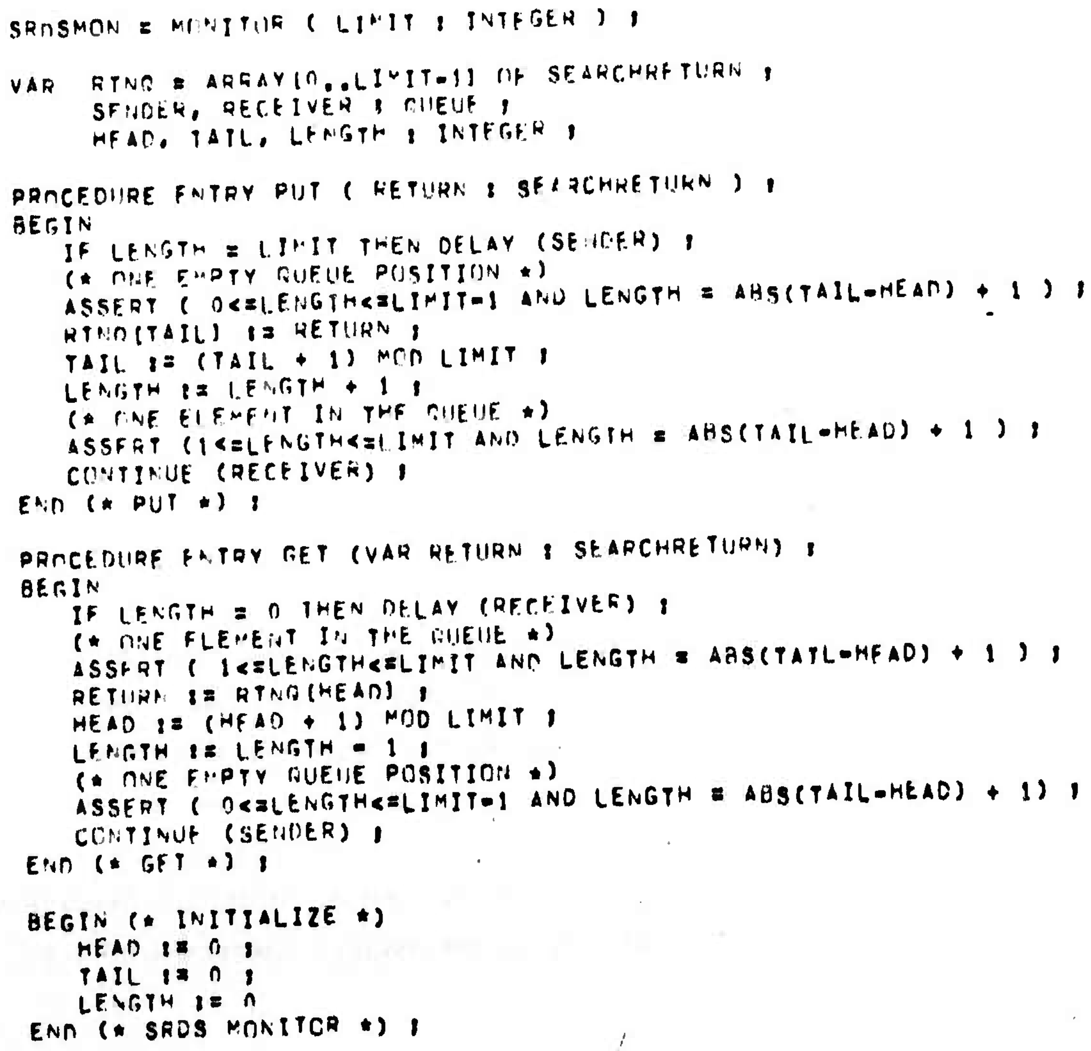
The final assertion in each monitor procedure must describe what result the execution of the procedure had upon the queue. For the PUT procedure, this is that an element is entered at the tail of the queue

RTNQ [ABS (HEAD - LENGTH)] = RETURN

Similarly in the GET procedure, the assertion must state that an element is retrieved from the head of the queue

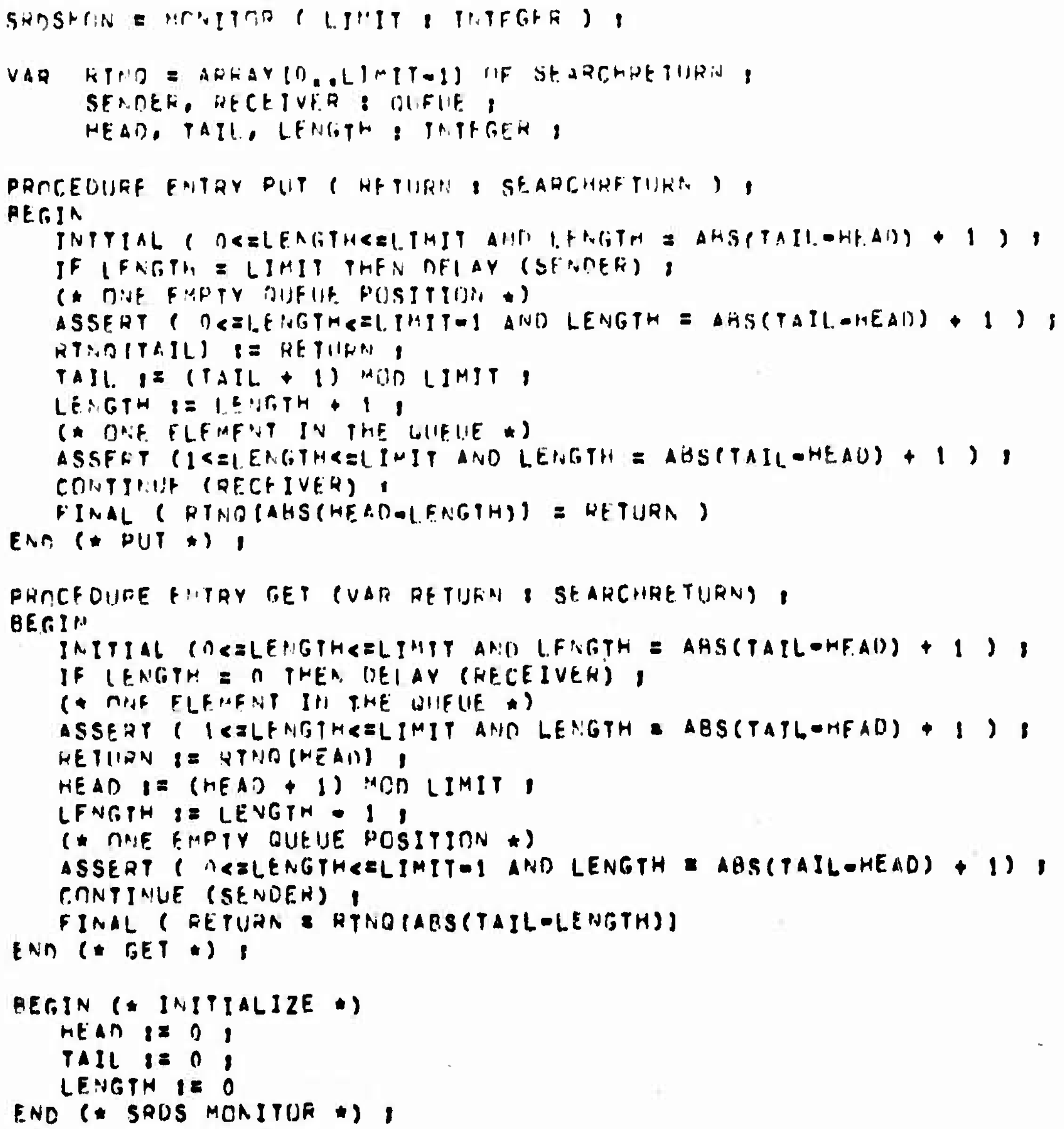
RETURN = RTNQ [ABS (TAIL - LENGTH)]

The monitor for the search return data set, complete with its assertions is shown in Fig. 7.13.



```
SRDSMON = MONITOR ( LIMIT : INTEGER ) ;

VAR  RTNQ : ARRAY[0..LIMIT-1] OF SEARCHRETURN ;
     SENDER, RECEIVER : QUEUE ;
     HEAD, TAIL, LENGTH : INTEGER ;

PROCEDURE ENTRY PUT ( RETURN : SEARCHRETURN ) ;
BEGIN
   IF LENGTH = LIMIT THEN DELAY (SENDER) ;
   (* ONE EMPTY QUEUE POSITION *)
   ASSERT ( 0<=LENGTH<=LIMIT-1 AND LENGTH = ABS(TAIL-HEAD) + 1 ) ;
   RTNQ[TAIL] := RETURN ;
   TAIL := (TAIL + 1) MOD LIMIT ;
   LENGTH := LENGTH + 1 ;
   (* ONE ELEMENT IN THE QUEUE *)
   ASSERT (1<=LENGTH<=LIMIT AND LENGTH = ABS(TAIL-HEAD) + 1 ) ;
   CONTINUE (RECEIVER) ;
END (* PUT *) ;

PROCEDURE ENTRY GET (VAR RETURN : SEARCHRETURN) ;
BEGIN
   IF LENGTH = 0 THEN DELAY (RECEIVER) ;
   (* ONE ELEMENT IN THE QUEUE *)
   ASSERT ( 1<=LENGTH<=LIMIT AND LENGTH = ABS(TATL-HEAD) + 1 ) ;
   RETURN := RTNQ[HEAD] ;
   HEAD := (HEAD + 1) MOD LIMIT ;
   LENGTH := LENGTH - 1 ;
   (* ONE EMPTY QUEUE POSITION *)
   ASSERT ( 0<=LENGTH<=LIMIT-1 AND LENGTH = ABS(TAIL-HEAD) + 1) ;
   CONTINUE (SENDER) ;
END (* GET *) ;

BEGIN (* INITIALIZE *)
   HEAD := 0 ;
   TAIL := 0 ;
   LENGTH := 0
END (* SRDS MONITOR *) ;
```

Figure 7.12. Assertions for DELAY and CONTINUE Statements



```
SRDSMON = MONITOR ( LIMIT : INTEGER ) ;

VAR  RTNQ = ARRAY[0..LIMIT-1] OF SEARCHRETURN ;
     SENDER, RECEIVER : QUEUE ;
     HEAD, TAIL, LENGTH : INTEGER ;

PROCEDURE ENTRY PUT ( RETURN : SEARCHRETURN ) ;
BEGIN
   INITIAL ( 0<=LENGTH<=LIMIT AND LENGTH = ABS(TAIL-HEAD) + 1 ) ;
   IF LENGTH = LIMIT THEN DELAY (SENDER) ;
   (* ONE EMPTY QUEUE POSITION *)
   ASSERT ( 0<=LENGTH<=LIMIT-1 AND LENGTH = ABS(TAIL-HEAD) + 1 ) ;
   RTNQ[TAIL] := RETURN ;
   TAIL := (TAIL + 1) MOD LIMIT ;
   LENGTH := LENGTH + 1 ;
   (* ONE ELEMENT IN THE QUEUE *)
   ASSERT (1<=LENGTH<=LIMIT AND LENGTH = ABS(TAIL-HEAD) + 1 ) ;
   CONTINUE (RECEIVER) ;
   FINAL ( RTNQ[ABS(HEAD-LENGTH)] = RETURN )
END (* PUT *) ;

PROCEDURE ENTRY GET (VAR RETURN : SEARCHRETURN) ;
BEGIN
   INITIAL (0<=LENGTH<=LIMIT AND LENGTH = ABS(TAIL-HEAD) + 1 ) ;
   IF LENGTH = 0 THEN DELAY (RECEIVER) ;
   (* ONE ELEMENT IN THE QUEUE *)
   ASSERT ( 1<=LENGTH<=LIMIT AND LENGTH = ABS(TAIL-HEAD) + 1 ) ;
   RETURN := RTNQ[HEAD] ;
   HEAD := (HEAD + 1) MOD LIMIT ;
   LENGTH := LENGTH - 1 ;
   (* ONE EMPTY QUEUE POSITION *)
   ASSERT ( 0<=LENGTH<=LIMIT-1 AND LENGTH = ABS(TAIL-HEAD) + 1) ;
   CONTINUE (SENDER) ;
   FINAL ( RETURN = RTNQ[ABS(TAIL-LENGTH)]
END (* GET *) ;

BEGIN (* INITIALIZE *)
   HEAD := 0 ;
   TAIL := 0 ;
   LENGTH := 0
END (* SRDS MONITOR *) ;
```

Figure 7.13. SRDS Monitor with Assertions

To generate verification conditions, we must perform symbolic execution over all paths between any two assertions. For all paths in the monitor except those containing DELAY statements, this is the same method as we have used for other programs. After a DELAY statement, however, any other monitor procedure may be executed. So, in the general case, we must generate verification conditions over all paths in all other monitors! Fortunately, we can use the procedure we have developed for subroutine calls to make this problem less difficult.

Since the verification condition generator and simplifier in the Software Quality Laboratory cannot as yet analyze Concurrent PASCAL, we will not show all the verification conditions and proofs for the SRDS monitor generated automatically. However, as a manually derived example, we will show that the "no blocking" criterion holds for this monitor. First assume that the PUT procedure has been called and that the queue is full. Therefore the process which called the PUT procedure is delayed and the condition

$$\text{LENGTH} = \text{LIMIT}$$

is true. The verification condition at the DELAY statement is therefore

$$0 \leq \text{LENGTH} = \text{ABS (TAIL - HEAD)} + 1 \leq \text{LIMIT}$$
$$\wedge \text{ LENGTH} = \text{LIMIT}$$

which simplifies to

$$0 \leq \text{LIMIT} = \text{ABS (TAIL - HEAD)} + 1 \leq \text{LIMIT}$$

and finally gives the expression

$$0 \leq \text{LIMIT} = \text{ABS (TAIL - HEAD)} + 1$$

Since there is only one sending process, the only monitor procedure which can be called next is the GET procedure. Examining the paths in the GET procedure we find that the process which invokes it can be delayed if the condition

$$\text{LENGTH} = 0$$

is true. The verification condition at the DELAY statement in the GET procedure is

$$0 \leq \text{LENGTH} = \text{ABS (TAIL - HEAD)} + 1 \leq \text{LIMIT} \wedge \text{LENGTH} = 0$$

which simplifies to

$$0 = \text{ABS (TAIL - HEAD)} = 1 \leq \text{LIMIT}$$

Both of these expressions must be true for each process to be writing on the other and since the expression ABS (TAIL - HEAD) + 1 appears in both verification conditions we obtain the combined expression

$$0 = \text{ABS (TAIL - HEAD)} + 1 = \text{LIMIT}$$

which implies that

$$\text{LIMIT} = 0$$

Since LIMIT is the length of the queue, the two processes cannot block each other unless the queue has a length of zero.

APPENDIX A

GRAMMAR DESCRIPTION FOR VERIFIABLE PASCAL

In the listing which follows is the syntactic description of Verifiable PASCAL as presented to the Compiler Writing System (CWS).[19] The imbedded semantic actions used to complete the Verifiable PASCAL Preprocessor have been removed from the data for clarity and brevity.

In preparing this grammar, close compatibility with the distributed version of the CDC 6000 standard PASCAL compiler was maintained. For example, the names of standard procedures and functions for that compiler are defined in the grammar, and are treated by the preprocessor as reserved identifiers.

In order to permit more effective error recovery in the generated preprocessor the grammar input to CWS departs slightly from the description shown in the syntax diagrams (see Sec. 3.2.1). For example, the rule for statement list <STALIST> does not require a semicolon to separate consecutive statements, but the syntax diagram does.

The grammar is stated as a set of rules in a form similar to BNF:

non-terminal = list of elements

As used in the grammar for Verifiable PASCAL, the elements are:

IDENT, a PASCAL identifier

ENTIER, a PASCAL integer

REEL, a PASCAL real number

CHAINE, a PASCAL character string

VIDE, an empty element

FDF, the end-of-file

≡ and ≡, delimiters for a reserved word or a reserved identifier

< and >, delimiters for a non-terminal identifier

V, to separate alternatives

[and], to group elements

[and], to signify that zero or more of the enclosed may occur

+[and]+, to signify that one or more of the enclosed occurs

=, to separate the left part of the rule for a non-terminal from the right part

$, to signify the end of a rule

The grammar is shown in Fig. A.1 preceded by a sequence of options for the first two phases of CWS. Rules of grammar begin after the symbol REGLE and terminate with the symbol FIN.

```
PROGRAMME ANALGEN.

DEBUT
PASCALV
OPTIONS
FONCTION
FTERMINA
GRAMMAIRE
REDUITE
LLUN
$
OPTIONS
ANGLAIS
MOTSCLES
OPERATEURS
RESERVES
PROCSYMBOLCAPTURE
BLANCSGROUPES
UTILIDENT 10
MAXIDENT  997
LONGZONECH4000
LONGCHAINE80
MAXNBCHIFF13
DELCOMMENT<*+>
$
TERMINAL
        IDENT
        ENTIER
        REEL
        CHAINE
        EOF
REGLE
<AXIOM> =
             <PROGRAM> EOF
             $
<PROGRAM> =
             ( ( ≡OPTIONS≡ <OPTIONLIST> ) v VIDE )
             ( ( ≡PROGRAM≡ <PROGHEAD> ) v VIDE )
             <BLOCK> ≡.≡
             $
<OPTIONLIST> =
             *( <SWITCH> ≡=≡ <OPTION> )*
             $
<SWITCH> =
             ≡ASSERT≡ v ≡UNITS≡ v ≡CONTROL≡ v ≡RIGHTS≡
             $
<OPTION> =
             ≡ON≡ v ≡OFF≡ v ≡ASIS≡
             $
<PROGHEAD> =
             IDENT <FILELIST> ≡;≡
             $
<FILELIST> =
             ≡(≡ <FILES> ≡)≡
             $
<FILES> =
             <FILEID> *( ≡,≡ <FILEID> )*
             $
<FILEID> =
             ( IDENT ( ≡*≡ v VIDE ) ) v ≡INPUT≡ v ≡OUTPUT≡
             $
```

Figure A.1. Grammar for Verifiable PASCAL

```
<BLOCK> =
          [ [ ≡LABEL≡ <LABELPART> ] v VIDE ]
          [ [ ≡CONST≡ <CONSTPART> ] v VIDE ]
          [ [ ≡TYPE≡ <TYPEPART> ] v VIDE ]
          [ [ ≡VAR≡ <VARPART> ] v VIDE ]
          [ *[ [ ≡PROCEDURE≡ <PROCPART> ] v [ ≡FUNCTION≡ <FUNCPART> ] ]* ]
          <BODY>
          $
<BODY> =
          ≡BEGIN≡ <STALIST> ≡END≡
          $
<LABELPART> =
          *[ ENTIER v ≡,≡ v ≡;≡ ]*
          $
<CONSTPART> =
          *[ <CONSTDEC> v ≡;≡ ]*
          $
<CONSTDEC> =
          IDENT ≡=≡ <CONSTANT> <UNITSDEC>
          $
<CONSTANT> =
          <CONSTA> v IDENT
          $
<CONSTA> =
          CHAINE v ≡TRUE≡ v ≡FALSE≡ v ≡MAXINT≡ v ≡NIL≡ v ENTIER v REEL
          v [ [ ≡+≡ v ≡-≡ ] [ ENTIER v REEL v IDENT ] ]
          $
<UNITSDEC> =
          [ ≡UNITS≡ <UNITSTERM> ] v VIDE
          $
<UNITSTERM> =
          <UNITSFACTOR> *[ [ ≡*≡ v ≡/≡ ] <UNITSFACTOR> ]*
          $
<UNITSFACTOR> =
          [ IDENT v ENTIER v REEL v [ ≡(≡ <UNITSTERM> ≡)≡ ] ]
          *[ ≡**≡ <UNITSPOWER> ]*
          $
<UNITSPOWER> =
          IDENT v ENTIER
          $
<TYPEPART> =
          *[ <TYPEDEC> v ≡;≡ ]*
          $
<TYPEDEC> =
          IDENT ≡=≡ <TYPE>
          $
<TYPE> =
          [ [ ≡PACKED≡ v VIDE ]
          [ [ ≡ARRAY≡ <ARRAYDEC> ]
          v [ ≡FILE≡ ≡OF≡ <TYPE> ]
          v [ ≡SEGMENTED≡ ≡FILE≡ ≡OF≡ <TYPE> ]
          v [ ≡SET≡ ≡OF≡ <SIMPLETYPE> ]
          v [ ≡RECORD≡ <FIELDLIST> ≡END≡ ] ] ]
          v [ ≡↑≡ IDENT ]
          v <SIMPLETYPE>
          $
<ARRAYDEC> =
          ≡[≡ <SIMPLETYPE> *[ ≡,≡ <SIMPLETYPE> ]* ≡]≡ ≡OF≡ <TYPE>
          $
```

Figure A.1 (Contd.)

```
<SIMFLETYPE> =
           [ ≡INTEGER≡ v ≡REAL≡ v ≡TEXT≡ v ≡CHAR≡ v ≡BOOLEAN≡ v ≡ALFA≡
           v [ ≡(≡ <IDLIST> ≡)≡ ]
           v [ <CONSTA> ≡..≡ <CONSTANT> ]
           v IDENT [ [ ≡..≡ <CONSTANT> ] v VIDE ] ]
           <UNITSDEC>
           $
<FIELCLIST> =
           [ <FIXEDPART> [ [ ≡CASE≡ <VARIANTPART> ] v VIDE ] ]
           v [ ( [ ≡CASE≡ <VARIANTPART> ] v VIDE ] ]
           $
<FIXECFAPT> =
           *[ <FIXEDDEC> v ≡;≡ ]*
           $
<FIXEDDEC> =
           <IDLIST> [ ≡:≡ v ≡%≡ ] <TYPE>
           $
<VARIAKTPART> =
           IDENT [ [ [ ≡:≡ v ≡%≡ ] IDENT ] v VIDE ] ≡OF≡ <CASE>
           *[ ≡CF≡ <CASE> ]*
           $
<CASE> =
           <CONSTANT> *[ ≡,≡ <CONSTANT> ]* [ ≡:≡ v ≡%≡ ] ≡(≡ <FIELDLIST> ≡)≡
           $
<VARPART> =
           *[ <VARDEC> v ≡;≡ ]*
           $
<VARDEC> =
           <IDLIST> [ ≡:≡ v ≡%≡ ] <TYPE>
           $
<IDLIST> =
           IDENT *[ ≡,≡ IDENT ]*
           $
<PRCCPART> =
           <PROCHEAD> ≡;≡ <PROCBODY> ≡;≡
           $
<FUNCFART> =
           <PROCHEAD> [ ≡:≡ v ≡%≡ ]
           [ IDENT v ≡INTEGER≡ v ≡BOOLEAN≡ v ≡REAL≡ v ≡CHAR≡ v ≡ALFA≡ ]
           ≡;≡ <PROCBODY> ≡;≡
           $
<PROCFEAD> =
           IDENT <PARAMLIST>
           $
<PRCCBCOY> =
           ≡FORTRAN≡ v ≡EXTERN≡ v ≡FORWARD≡ v <BLOCK>
           $
<FARAMLIST> =
           [ ≡(≡ <PARAMS>  ≡)≡ ] v VIDE
           $
<PARAMS> =
           <PARAM> *[ ≡;≡ <PARAM> ]*
           $
<PARAM> =
           [ ≡PROCEDURE≡ <IDLIST> ]
           v [ [ ≡FUNCTION≡ v ≡VAR≡ v VIDE ] <IDLIST> [ ≡:≡ v ≡%≡ ]
           [ IDENT v ≡INTEGER≡ v ≡BOOLEAN≡ v ≡REAL≡ v ≡CHAR≡ v ≡ALFA≡ v ≡TEXT≡ ] ]
           $
<STALIST> =
```

Figure A.1 (Contd.)

```
            *[ <STATEMENT> v ≡;≡ ]*
            $
<STATEMENT> =
            <STALABEL> v <STATYPE>
            $
<STALABEL> =
            ENTIER [ ≡:≡ v ≡%≡ ]
            $
<STATYFE> =
            [ ≡BEGIN≡ <STALIST> ≡END≡ ]
            v [ ≡IF≡ <IFSTA> ]
            v [ ≡CASE≡ <CASESTA> ]
            v [ ≡WHILE≡ <WHILESTA> ]
            v [ ≡REPEAT≡ <REPEATSTA> ]
            v [ ≡FOR≡ <FORSTA> ]
            v [ ≡WITH≡ <WITHSTA> ]
            v [ ≡GOTO≡ <GOTOSTA> ]
            v [ [ IDENT v <STDPROC> ] [ [ ≡(≡ <INVLIST> ≡)≡ ]
            v [ <QUALLIST> [ ≡:=≡ v ≡%=≡ ] <EXPRESSION> ]
            v [ [ ≡:=≡ v ≡%=≡ ] <EXPRESSION> ] v VIDE ] ]
            v [ ≡ASSERT≡ <ASSERTSTA> ]
            v [ ≡INITIAL≡ <ASSERTSTA> ]
            v [ ≡FINAL≡ <ASSERTSTA> ]
            v [ ≡INPUT≡ <IDLIST> ]
            v [ ≡OUTPUT≡ <IDLIST> ]
            $
<IFSTA> =
            <CONDITIONAL>
            *[ ≡ORIF≡ <CONDITIONAL> ]*
            [ [ ≡ELSE≡ <STALIST> ] v VIDE ]
            [ ≡ENDIF≡ v [ ≡END≡ ≡IF≡ ] ]
            $
<CONDITIONAL> =
            <EXPRESSION> ≡THEN≡ <STALIST>
            $
<CASESTA> =
            <EXPRESSION> ≡OF≡ [ <CASP> v VIDE ]
            *[ ≡OF≡ [ <CASP> v VIDE ] ]*
            [ ≡ENDCASE≡ v [ ≡END≡ ≡CASE≡ ] ]
            $
<CASP> =
            <CASELABEL> <STALIST>
            $
<CASELABEL> =
            <CONSTANT> *[ ≡,≡ <CONSTANT> ]* [ ≡:≡ v ≡%≡ ]
            $
<WHILESTA> =
            <EXPRESSION> ≡DO≡ <STALIST>
            [ ≡ENDWHILE≡ v [ ≡END≡ ≡WHILE≡ ] ]
            $
<REPEATSTA> =
            <STALIST> ≡UNTIL≡ <EXPRESSION>
            $
<FORSTA> =
            IDENT [ ≡:=≡ v ≡%=≡ ] <FORLIST> ≡DO≡ <STALIST>
            [ ≡ENDFOR≡ v [ ≡END≡ ≡FOR≡ ] ]
            $
<FORLIST> =
            <EXPRESSION> [ ≡TO≡ v ≡DOWNTO≡ ] <EXPRESSION>
```

Figure A.1 (Contd.)

```
          $
<WITHSTA> =
          <VARLIST> ≡DO≡ <STALIST>
          [ ≡ENDWITH≡ ∨ [ ≡END≡ ≡WITH≡ ] ]
          $
<VARLIST> =
          <VARIABLE> *[ ≡,≡ <VARIABLE> ]*
          $
<VARIABLE> =
          IDENT [ <QUALLIST> ∨ VIDE ]
          $
<QUALLIST> =
          +[ [ ≡[≡ <EXPRESSION> *[ ≡,≡ <EXPRESSION> ]* ≡]≡ ]
          ∨ [ ≡.≡ IDENT ]
          ∨ ≡↑≡ ]+
          $
<GOTOSTA> =
          ENTIER
          $
<ASSERTSTA> =
          <ASSERTEXP> [ [ ≡FAIL≡ <STALIST>
          [ ≡ENDFAIL≡ ∨ [ ≡END≡ ≡FAIL≡ ] ] ] ∨ VIDE ]
          $
<ASSERTEXP> =
          <QUANTEXP> [ [ ≡=>≡ <QUANTEXP> ] ∨ VIDE ]
          $
<QUANTEXP> =
          [ <EXPRESSION> ] ∨ [ [ ≡ALL≡ ∨ ≡SOME≡ ] <QUANTAIL> ]
          $
<QUANTAIL> =
          ≡(≡ IDENT ≡IN≡ <QUANTLIST> ≡IS≡ <ASSERTEXP> ≡)≡
          $
<QUANTLIST> =
          <EXPRESSION> [ ≡TO≡ ∨ ≡DOWNTO≡ ] <EXPRESSION>
          $
<INVLIST> =
          <INVPAR> *[ ≡,≡ <INVPAR> ]*
          $
<INVPAR> =
          [ <EXPRESSION> *[ [ ≡:≡ ∨ ≡%≡ ] <EXPRESSION> ]* ]
          ∨ ≡INPUT≡ ∨ ≡OUTPUT≡
          $
<EXPRESSION> =
          <SIMPLEEXP>
          [ [ [ ≡=≡ ∨ ≡<≡ ∨ ≡>≡ ∨ ≡≠≡ ∨ ≡≤≡ ∨ ≡≥≡ ∨ ≡<>≡ ∨ ≡<=≡ ∨ ≡>=≡
          ∨ ≡EQ≡ ∨ ≡LE≡ ∨ ≡NE≡ ∨ ≡GE≡ ∨ ≡LT≡ ∨ ≡GT≡ ∨ ≡IN≡ ]
          <SIMPLEEXP> ] ∨ VIDE ]
          $
<SIMPLEEXP> =
          [ [ ≡+≡ <TERM> ] ∨ [ ≡-≡ <TERM> ] ∨ <TERM> ]
          *[ [ ≡+≡ ∨ ≡-≡ ∨ ≡OR≡ ∨ ≡∨≡ ] <TERM> ]*
          $
<TERM> =
          <FACTOR> *[ [ ≡*≡ ∨ ≡/≡ ∨ ≡DIV≡ ∨ ≡MOD≡ ∨ ≡AND≡ ∨ ≡∧≡ ]
          <FACTOR> ]*
          $
<FACTOR> =
          [ ≡[≡ [ [ <SET> ≡]≡ ] ∨ ≡]≡ ] ]
          ∨ [ [ ≡NOT≡ ∨ ≡¬≡ ] <FACTOR> ]
```

Figure A.1 (Contd.)

```
           v [ ≡(≡ <EXPRESSION> ≡)≡ ]
           v [ [ IDENT v <STDFUNC> ]
           [ [ ≡(≡ <INVLIST> ≡)≡ ] v <QUALLIST> v VIDE ] ]
           v CHAINE v REEL v ENTIER v ≡NIL≡ v ≡TRUE≡ v ≡FALSE≡ v ≡MAXINT≡
           $
<SET> =
           <MEMBER> *[ ≡,≡ <MEMBER> ]*
           $
<MEMBER> =
           <EXPRESSION> [ [ ≡..≡ <EXPRESSION> ] v VIDE ]
           $
<STDPRCC> =
           ≡GFT≡ v ≡NEW≡ v ≡PACK≡ v ≡PAGE≡ v ≡PUT≡ v ≡READ≡ v ≡READLN≡
           v ≡RESET≡ v ≡REWRITE≡ v ≡UNPACK≡ v ≡WRITE≡ v ≡WRITELN≡
           v ≡DATE≡ v ≡GETSEG≡ v ≡HALT≡ v ≡LINELIMIT≡ v ≡MESSAGE≡
           v ≡PUTSEG≡ v ≡TIME≡ v ≡DISPOSE≡ v ≡RELEASE≡
           $
<STDFUNC> =
           ≡ABS≡ v ≡ARCTAN≡ v ≡CHR≡ v ≡COS≡ v ≡EOF≡ v ≡EOLN≡ v ≡EXP≡
           v ≡LN≡ v ≡ODD≡ v ≡ORD≡ v ≡PRED≡ v ≡ROUND≡ v ≡SIN≡ v ≡SQR≡
           v ≡SQRT≡ v ≡SUCC≡ v ≡TRUNC≡
           v ≡CARD≡ v ≡CLOCK≡ v ≡EOS≡ v ≡EXPO≡ v ≡UNDEFINED≡ v ≡RANDOM≡
           $
FIN
```

Figure A.1 (Contd.)

APPENDIX B

TRANSLATION TEMPLATES FOR VERIFIABLE PASCAL

The templates for translating Verifiable PASCAL control structures and executable assertions are outlined below. Where appropriate, keywords are retained in the translated text as comments (e.g., ENDIF becomes (*ENDIF*)). Original comments are suppressed.

For the option ASSERT = ON, the following variable declaration statement is inserted into the VAR group of each module:

```
ASSERT : BOOLEAN
```

If a module has no VAR group, one is inserted automatically.

For a Boolean function which contains only assertion statements, an assignment statement is generated at the end of the statement list for the value of the function:

```
function name := ASSERT
```

The templates shown below are valid only for an assertion expression which does not contain another assertion expression as a subexpression.

Statement	Source	Translated Source
IF	IF expression THEN	IF expression THEN
	statement list	BEGIN statement list
	ORIF expression THEN	END (*ORIF*)
	statement list	ELSE IF expression THEN
		BEGIN statement list
	ELSE	END
	statement list	ELSE
		BEGIN statement list
	ENDIF	END (*ENDIF*)
CASE	CASE expression OF	CASE expression OF
	case label list:	
	statement list	BEGIN statement list END
	OF case label list:	;(*OF*) case label list:
	statement list	BEGIN statement list END
	END CASE	END (*END CASE*)
WHILE	WHILE expression DO	WHILE expression DO
	statement list	BEGIN statement list
	END WHILE	END (*END WHILE*)
FOR	FOR control variable	FOR control variable
	:= for list DO	:= for list DO
	statement list	BEGIN statement list
	END FOR	END (*END FOR*)
WITH	WITH variable list DO	WITH variable list DO
	statement list	BEGIN statement list
	END WITH	END (*END WITH*)
INITIAL,	INITIAL	(*keyword*) assertion expression
ASSERT,	ASSERT assertion expression	code;
FINAL	FINAL	
	(without fail clause)	IF NOT ASSERT THEN
		WRITELN ('ASSERT FALSE
		AT TEXT LINE statement
		number FOR STATEMENT
		AT module statement')
	FAIL statement list	IF NOT ASSERT THEN
	END FAIL	(*FAIL*) BEGIN
		statement list
		END (*END FAIL*)

	<u>first order expression</u>	
assertion expression*	p AND q	ASSERT := p AND q
	p OR q	ASSERT := p OR q

	p => q	ASSERT :=p;
		IF ASSERT THEN
		  BEGIN
		  ASSERT := q
		  END
		ELSE
		  ASSERT := TRUE

	ALL (i IN m TO n IS p)	ASSERT := TRUE; i := m;
		WHILE (i <= n) AND ASSERT DO
		BEGIN ASSERT := p;
		i := i + 1 END

	ALL (i IN m DOWNTO n IS p)	ASSERT := TRUE; i := m;
		WHILE (i >= n) AND ASSERT DO
		BEGIN ASSERT := p;
		i := i - 1 END

	SOME (i IN m TO n IS p)	ASSERT := FALSE; i := m;
		WHILE (i <= n) AND NOT ASSERT DO
		BEGIN ASSERT := p;
		i := i + 1 END

	SOME (i IN m DOWNTO n IS p)	ASSERT := FALSE; i := m;
		WHILE (i >= n) AND NOT ASSERT DO
		BEGIN ASSERT := p;
		i := i - 1 END

*Templates are valid for only where p and q are PASCAL expressions of Boolean type, i is a PASCAL integer variable, and m and n are PASCAL expressions of integer type.

APPENDIX C

TRANSLATION TEMPLATES FOR IFTRAN WITH ASSERTIONS

The IFTRAN preprocessor recognizes a set of assertion statements which can be translated to compilable FORTRAN and executed. The default translation of IFTRAN assertions is into FORTRAN comments. This section describes the IFTRAN commands which cause executable assertion translation.

The executable form of the INPUT and OUTPUT statements writes the current values of all variables in their variable lists. Thus, the INPUT (OUTPUT) statement prints input (output) variables of a routine when used as described in Sec. 3.2. The commands associated with INPUT and OUTPUT statements are

IFTRAN COMMAND	FUNCTION
TRAC	Trace input and output values
TROF	Resume default mode of no tracing
UNIT name	Use as output unit the FORTRAN variable or constant name

Figure C.1 is an example of a subroutine which uses the trace commands to indicate its input and output values whenever it is executed.

The executable form of INITIAL, FINAL, and ASSERT statements evaluates the first-order logic expression asserted with current values of program variables. If the expression is false, an error message is printed and any associated FAIL BLOCK is executed. The syntax of INITIAL, FINAL, and ASSERT statements with or without FAIL clauses is defined in Sec. 3.3. The commands associated with ENTRY, EXIT, and ASSERT statements are

IFTRAN COMMAND	FUNCTION
ASON	Check assertions for validity during execution
ASOF	Resume default mode of no checking
UNIT name	Use the FORTRAN variable or constant NAME as output unit
MODN name	Provides the name of a routine which has ENTRY, EXIT, or ASSERT statements

The "ASON" and "ASOF" and "UNIT" commands are global commands in that they refer to more than one routine. The "MODN" command is local to an individual routine. It can occur anywhere that a logical declaration statement is legal in FORTRAN. Deck specific executable assertion checking is turned on by the "MODN" command. An example of a subroutine which will produce exception reports when executed is given in Fig. C.2

The executable code for each type of assertion expression is given in Fig. C.3.

```
      SUBROUTINE MLTPLY(A,B,C,N)
CTRAC
CUNIT LOUT
      DIMENSION A(1,1),B(1,1),C(1,1)
      DATA LOUT /6/
      INPUT(N,A,B)
      DO(I=1,N)
      .   DO(J=1,N)
      .   .   INVOKE( COMPUTE NEW ARRAY ELEMENT )
      .   END DO
      END DO

      BLOCK( COMPUTE NEW ARRAY ELEMENT )
      .   S = 0.
      .   DO(K=1,N)
      .   .   S = S + A(I,K) * B(K,J)
      .   END DO
      .   C(I,J) = S
      END BLOCK
      OUTPUT(C)
      RETURN
      END
```

Figure C.1. Trace Commands

```
      FUNCTION SQRT(A)
CASON
CMODN SQRT
CUNIT LOUT
      DATA LOUT /6/
      INITIAL( A .GT. 0 )
      X = A
      WHILE( ABS(X-A/X) .GT. 1.E-6 )
      .   X = (X + A/X)/2
      END WHILE
      SQRT = X
      FINAL( ABS(SQRT**2 - A ) .LE. 1.E-6 )
      RETURN
      END
```

Figure C.2. Executable Assertion Commands

```
ASSERTION EXPRESSION

(P .AND. Q)                    ASSERT = P
                               IF (ASSERT)
                                  ASSERT = Q
                               ENDIF

(P .OR. Q)                     ASSERT = P
                               IF (.NOT. ASSERT)
                                  ASSERT = Q
                               ENDIF

(P .IMP. Q)                    ASSERT = P
                               IF (ASSERT)
                                  ASSERT = Q
                               ELSE
                                  ASSERT = .TRUE.
                               ENDIF

(ALL I IN (1,N) (P))           ASSERT = .TRUE.
                               I=1
                               WHILE (I .LE. N .AND. ASSERT)
                                  ASSERT = P
                                  IF (ASSERT) I = I + 1
                               ENDWHILE

(SOME I IN (1,N) (P))          I=1
                                  (I .GT. N .OR. ASSERT)
                                  ASSERT = P
                                  IF (.NOT. ASSERT) I = I + 1
                               UNTIL
                                  (I .GT. N .OR. ASSERT)
```

Figure C.3. Translation Templates for IFTRAN Assertions

APPENDIX D

VERIFIED PROGRAMS

```
STATEMENT LISTING                   SUBROUTINE TIMES ( A, B, X )                              PAGE      1

  NO. LEVEL     LABEL   STATEMENT TEXT...                                                       CDPATHS
------------------------------------------------------------------------------------------------------------
    1                   SUBROUTINE TIMES ( A, B, X )                                          (  1)
    2             C     MODNAM(TIMES)
    3             C     KING S EXAMPLE 1
    4             C
    5             C     USE ADDITION TO PERFORM MULTIPLICATION OF A TIMES B
    6             C     THE RESULT IS IN X
    7             C
    8                   INTEGER A, B, X
    9                   INTEGER Y, SUM
   10             C
   11                   INITIAL ( B .GE. 0 )
   12                   SUM = 0
   13                   Y = B
   14             C
   15                   WHILE ( Y .NE. 0 )                                                    (  2-  3)
   16 ( 1)              .  SUM = SUM + A
   17 ( 1)              .  Y = Y - 1
   18 ( 1)              .  ASSERT ( SUM .EQ. A * ( B - Y ) .AND. Y .GE. 0 )
   19                   ENDWHILE
   20                   X = SUM
   21                   FINAL ( X .EQ. A * B )
   22                   RETURN
   23                   END
------------------------------------------------------------------------------------------------------------
```

```
STATEMENT LISTING                  SUBROUTINE DIV ( A, B, Q, R )                        PAGE      1

 NO. LEVEL    LABEL   STATEMENT TEXT...                                                 DDPATHS
---------------------------------------------------------------------------------------------------
    1                 SUBROUTINE DIV ( A, B, Q, R )                                     (  1)
    2           C     ROLN DIV
    3           C     KING S EXAMPLE 2
    4           C
    5           C     USE SUBTRACTION TO PERFORM DIVISION OF A BY B
    6           C     THE QUOTIENT IS IN Q WITH THE REMAINDER IN R
    7           C
    8                 INTEGER A, B, Q, R
    9           C
   10                 INITIAL ( A .GE. 0 .AND. B .GE. 0 )
   11                 Q = 0
   12                 R = A
   13                 WHILE ( R .GE. B )                                                (  2-  3)
   14 ( 1)            .  ASSERT ( A .EQ. Q * B + R .AND. R .GE. 0 )
   15 ( 1)            .  Q = Q + 1
   16 ( 1)            .  R = R - B
   17                 ENDWHILE
   18                 FINAL ( A .EQ. Q * B + R .AND. 0 .LE. R .AND. R .LT. B )
   19                 RETURN
   20                 END
---------------------------------------------------------------------------------------------------
```

```
STATEMENT LISTING                  SUBROUTINE EXPON ( A, B, Z )                              PAGE     1

 NO. LEVEL     LABEL   STATEMENT TEXT...                                                    CDPATHS
-------------------------------------------------------------------------------------------------------
    1                  SUBROUTINE EXPON ( A, B, Z )                                        (  1)
    2            C     MODE EXPON
    3                  INTEGER A, B, Z, X, Y
    4            C     KINGS EXAMPLE 3 EXPONENTIATION BY MULTIPLICATION
    5                  INITIAL ( B .GE. 0 )
    6                  X = A
    7                  Y = B
    8                  Z = 1
    9                  WHILE ( Y .NE. 0 )                                                  (  2-  3)
   10 ( 1)             .  ASSERT ( Y .GE. 0 .AND. Z * ( X ** Y ) .EQ. A ** B )
   11 ( 1)             .  IF ( MOD ( Y, 2 ) .EQ. 1 )                                       (  4-  5)
   12 ( 2)             .  .  Z = Z * X
   13 ( 1)             .  ENDIF
   14 ( 1)             .  Y = Y / 2
   15 ( 1)             .  X = X * X
   16                  ENDWHILE
   17                  FINAL ( Z .EQ. A ** B )
   18                  RETURN
   19                  END
-------------------------------------------------------------------------------------------------------
```

```
STATEMENT LISTING                     SUBROUTINE PRIME ( A, J )                                                  PAGE      1

  NO. LEVEL     LABEL   STATEMENT TEXT...                                                                        DDPATHS
--------------------------------------------------------------------------------------------------------------------------
    1                   SUBROUTINE PRIME ( A, J )                                                                 (  1)
    2             C     MOON PRIME
    3             C     KINGS EXAMPLE 4
    4             C     IF A IS PRIME, J IS SET TO 0
    5             C     IF A IS NOT PRIME, J IS SET TO 1
    6                   INTEGER A, I, J, K
    7                   INITIAL ( A .GE. 2 )
    8                   I = 2
    9                   WHILE ( I .LT. A .AND. MOD ( A, I ) .NE. 0 )                                              (  2-   3)
   10 ( 1)              .  ASSERT ( ( .ALL. K .IN. ( 2, I ) ( MOD ( A, K ) .NE. 0 ) ) .AND. I
                       *.    .LE. A )
   11 ( 1)              .  I = I + 1
   12                   ENDWHILE
   13                   IF ( 1 .NE. A )                                                                           (  4-   5)
   14 ( 1)              .  J = 1
   15                   ELSE
   16 ( 1)              .  J = 0
   17                   ENDIF
   18                   FINAL ( ( J .EQ. 0 .IMP. ( .ALL. K .IN. ( 2, A - 1 ) ( MOD ( A, K
                       *) .NE. 0 ) ) ) .AND. ( J .EQ. 1 .IMP. MOD ( A, I ) .EQ. 0 ) )
   19                   RETURN
   20                   END
--------------------------------------------------------------------------------------------------------------------------
```

```
STATEMENT LISTING                  SUBROUTINE ZERO ( A, N )                                            PAGE      1
                                                                                                     
  NO. LEVEL     LABEL   STATEMENT TEXT...                                                            CDPATHS
-----------------------------------------------------------------------------------------------------------------
     1                   SUBROUTINE ZERO ( A, N )                                                    (  1)
     2            C      MOUNAM (ZERO)
     3            C
     4            C      KINGS EXAMPLE 5
     5            C      SET AN ARRAY A OF LENGTH N TO ZERO
     6                   INTEGER A ( 1 ), N, I, J
     7                   INITIAL ( .TRUE. )
     8                   I = 1
     9                   WHILE ( I .LE. N )                                                          (  2-  3)
    10 ( 1)              .  A ( I ) = 0
    11 ( 1)              .  ASSERT ( ( .ALL. J .IN. ( 1, I - 1 ) ( A ( J ) .EQ. 0 ) ) .AND. A
                        *.  ( I ) .EQ. 0 )
    12 ( 1)              .  I = I + 1
    13                   ENDWHILE
    14                   FINAL ( .ALL. J .IN. ( 1, N ) ( A ( J ) .EQ. 0 ) )
    15                   RETURN
    16                   END
-----------------------------------------------------------------------------------------------------------------
```

```
STATEMENT LISTING                SUBROUTINE MAXI ( A, N )                                              PAGE     1

 NO. LEVEL    LABEL   STATEMENT TEXT...                                                                  DDPATHS
-----------------------------------------------------------------------------------------------------------------
   1                  SUBROUTINE MAXI ( A, N )                                                          (  1)
   2            C     MODNAM (MAXI)
   3            C     KINGS EXAMPLE 6  STORES MAX VALUE IN A IN A(N)
   4                  INTEGER A ( 1 ), N, I, K, TEMP
   5                  INITIAL ( N .GT. 0 )
   6                  I = 2
   7                  WHILE ( I .LE. N )                                                                (  2-  3)
   8 ( 1)             .  IF ( A ( I - 1 ) .GT. A ( I ) )                                                (  4-  5)
   9 ( 2)             .  .  TEMP = A ( I )
  10 ( 2)             .  .  A ( I ) = A ( I - 1 )
  11 ( 2)             .  .  A ( I - 1 ) = TEMP
  12 ( 1)             .  ENDIF
  13 ( 1)             .  ASSERT ( ( .ALL. K .IN. ( 1, I - 1 ) ( A ( I ) .GE. A ( K ) ) )
                     *.  .AND. I .LE. N )
  14 ( 1)             .  I = I + 1
  15                  ENDWHILE
  16                  FINAL ( .ALL. K .IN. ( 1, N - 1 ) ( A ( N ) .GE. A ( K ) ) )
  17                  RETURN
  18                  END
-----------------------------------------------------------------------------------------------------------------
```

```
STATEMENT LISTING                  SUBROUTINE SORT1 ( A, N )                                    PAGE      1

  NO. LEVEL     LABEL   STATEMENT TEXT...                                                           DDPATHS
---------------------------------------------------------------------------------------------------------------
   1                    SUBROUTINE SORT1 ( A, N )                                                ( 1)
   2             C      MODNAM (SORT1)
   3             C
   4             C      KINGS EXAMPLE 7
   5             C      SORT AN ARRAY OF LENGTH N
   6             C
   7                    INTEGER A ( 1 ), N
   8                    INTEGER I, J, M, TEMP
   9             C
  10                    INITIAL ( .TRUE. )
  11                    J = 1
  12                    WHILE ( J .EQ. 1 )                                                       (  2-  3)
  13 ( 1)               .  J = 0
  14 ( 1)               .  I = 2
  15 ( 1)               .  WHILE ( I .LE. N )                                                    (  4-  5)
  16 ( 2)               .  .  ASSERT ( .ALL. L .IN. ( 2, I - 1 ) ( J .NE. 0 .OR. A ( L - 1 )
                       *.        .LE. A ( L ) ) )
  17 ( 2)               .  .  IF ( A ( I - 1 ) .GE. A ( I ) )                                    (  6-  7)
  18 ( 3)               .  .  .  TEMP = A ( I - 1 )
  19 ( 3)               .  .  .  A ( I - 1 ) = A ( I )
  20 ( 3)               .  .  .  A ( I ) = TEMP
  21 ( 3)               .  .  .  J = 1
  22 ( 2)               .  .  ENDIF
  23 ( 2)               .  .  I = I + 1
  24 ( 1)               .  ENDWHILE
  25                    ENDWHILE
  26                    FINAL ( .ALL. M .IN. ( 2, N ) ( A ( M - 1 ) .LE. A ( M ) ) )
  27                    RETURN
  28                    END
---------------------------------------------------------------------------------------------------------------
```

```
STATEMENT LISTING                   SUBROUTINE MULT2 ( A, B, ANS )                                        PAGE      1

  NO. LEVEL     LABEL   STATEMENT TEXT...                                                                  CDPATHS
---------------------------------------------------------------------------------------------------------------------
    1                   SUBROUTINE MULT2 ( A, B, ANS )                                                    (  1)
    2             C     MODNAM(MULT2)
    3             C     KINGS EXAMPLE 8
    4             C     MULTIPLY A TIMES B, RESULT IN ANS
    5             C
    6                   INTEGER A, B, TANS
    7                   INTEGER ANS, TEMPA, TEMPB
    8                   INITIAL ( .TRUE. )
    9                   TEMPA = A
   10                   TANS = 0
   11                   WHILE ( TEMPA .NE. 0 )                                                            (  2-  3)
   12 ( 1)              .  ASSERT ( TANS .EG. ( A - TEMPA ) * B )
   13 ( 1)              .  TEMPB = B
   14 ( 1)              .  IF ( TEMPA .GT. 0 )                                                            (  4-  5)
   15 ( 2)              .  .  WHILE ( TEMPB .NE. 0 )                                                      (  6-  7)
   16 ( 3)              .  .  .  IF ( TEMPB .GT. 0 )                                                      (  8-  9)
   17 ( 4)              .  .  .  .  TANS = TANS + 1
   18 ( 4)              .  .  .  .  TEMPB = TEMPB - 1
   19 ( 3)              .  .  .  ELSE
   20 ( 4)              .  .  .  .  TANS = TANS - 1
   21 ( 4)              .  .  .  .  TEMPB = TEMPB + 1
   22 ( 3)              .  .  .  ENDIF
   23 ( 3)              .  .  .  ASSERT ( TANS .EG. ( A - TEMPA ) * B + B - TEMPB )
   24 ( 2)              .  .  ENDWHILE
   25 ( 2)              .  .  TEMPA = TEMPA - 1
   26 ( 1)              .  ELSE
   27 ( 2)              .  .  WHILE ( TEMPB .NE. 0 )                                                      ( 10- 11)
   28 ( 3)              .  .  .  IF ( TEMPB .GT. 0 )                                                      ( 12- 13)
   29 ( 4)              .  .  .  .  TANS = TANS - 1
   30 ( 4)              .  .  .  .  TEMPB = TEMPB - 1
   31 ( 3)              .  .  .  ELSE
   32 ( 4)              .  .  .  .  TANS = TANS + 1
   33 ( 4)              .  .  .  .  TEMPB = TEMPB + 1
   34 ( 3)              .  .  .  ENDIF
   35 ( 3)              .  .  .  ASSERT ( TANS .EG. ( A - TEMPA ) * B - B + TEMPB )
   36 ( 2)              .  .  ENDWHILE
   37 ( 2)              .  .  TEMPA = TEMPA + 1
   38 ( 1)              .  ENDIF
   39                   ENDWHILE
   40                   ANS = TANS
   41                   FINAL ( ANS .EG. A * B )
   42                   RETURN
   43                   END
---------------------------------------------------------------------------------------------------------------------
```

```
STATEMENT LISTING                    SUBROUTINE SORT2 ( A, N )                              PAGE      1
  NO. LEVEL    LABEL   STATEMENT TEXT...                                                       DDPATHS
---------------------------------------------------------------------------------------------------------
    1                  SUBROUTINE SORT2 ( A, N )                                               (  1)
    2           C      ROUMAP(SORT2)
    3           C      KINGS EXAMPLE 9
    4                  INTEGER A ( 1 ), N
    5                  INTEGER I, J, K, L, M, TEMP
    6                  INITIAL ( .TRUE. )
    7                  I = 1
    8                  WHILE ( I .LT. N )                                                      (  2-  3)
    9 ( 1)      C      .  I INVARIANT
   10 ( 1)             .  ASSERT ( I .LE. N .AND. ( .ALL. L .IN. ( 2, I - 1 ) ( A ( L - 1 )
                     *.  .LE. A ( L ) ) ) )
   11 ( 1)             .  J = I + 1
   12 ( 1)             .  TEMP = A ( I )
   13 ( 1)             .  K = I
   14 ( 1)             .  WHILE ( J .LE. N )                                                   (  4-  5)
   15 ( 2)      C      .  .  J INVARIANT
   16 ( 2)             .  .  ASSERT ( J .LE. N + 1 .AND. I .LE. K .AND. K .LT. J .AND. TEMP
                     *.     .EQ. A ( K ) .AND. ( .ALL. L .IN. ( 2, I - 1 ) ( A ( L - 1 ) .LE.
                     *.     A ( L ) ) ) .AND. ( .ALL. M .IN. ( I, J - 1 ) ( A ( M ) .LE. A ( K
                     *.       ) ) ) )
   17 ( 2)             .  .  IF ( TEMP .GT. A ( J ) )                                          (  6-  7)
   18 ( 3)             .  .  .  TEMP = A ( J )
   19 ( 3)             .  .  .  K = J
   20 ( 2)             .  .  ENDIF
   21 ( 2)             .  .  J = J + 1
   22 ( 1)             .  ENDWHILE
   23 ( 1)      C      .  J INVARIANT
   24 ( 1)             .  ASSERT ( J .LE. N + 1 .AND. I .LE. K .AND. K .LT. J .AND. TEMP
                     *.  .EQ. A ( K ) .AND. ( .ALL. L .IN. ( 2, I - 1 ) ( A ( L - 1 ) .LE.
                     *.  A ( L ) ) ) .AND. ( .ALL. M .IN. ( 1, J - 1 ) ( A ( M ) .LE. A ( K
                     *.    ) ) ) )
   25 ( 1)             .  A ( K ) = A ( I )
   26 ( 1)             .  A ( I ) = TEMP
   27 ( 1)             .  I = I + 1
   28                  ENDWHILE
   29                  FINAL ( .ALL. L .IN. ( 2, N ) ( A ( L - 1 ) .LE. A ( L ) ) )
   30                  RETURN
   31                  END
---------------------------------------------------------------------------------------------------------
```

```
STATEMENT LISTING                    SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )            PAGE       1

 NO. LEVEL    LABEL   STATEMENT TEXT...                                                                 COPATHS
------------------------------------------------------------------------------------------------------------------
    1                 SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )                            (  1)
    2          C      MODNAM(BINSCH)
    3          C
    4          C      CONDUCT A BINARY SEARCH BY SUCCESSIVE HALVING OF A SORTED ARRAY
    5          C
    6          C      SEARCH THE ARRAY TO FIND THE ELEMENT WITH VALUE X
    7          C
    8          C      IF X IS FOUND, SET THE BOOLEAN FLAG VARIABLE ERROR TO FALSE AND
    9          C      THE INDEX LOOKUP TO THE ELEMENT  WHERE X WAS FOUND
   10          C      IF X IS NOT FOUND, SET THE BOOLEAN FLAG VARIABLE TO TRUE
   11          C
   12                 INTEGER ARRAY ( 1 ), LENGTH, X, LOOKUP
   13                 LOGICAL ERROR, SORTED
   14                 INTEGER I, M, N
   15                 INITIAL ( 1 .LT. LENGTH .AND. SORTED ( ARRAY, LENGTH ) .AND. (
                     *ARRAY ( 1 ) .LE. X .AND. X .LT. ARRAY ( LENGTH ) ) )
   16                 M = 1
   17                 N = LENGTH
   18                 ERROR = .FALSE.
   19                 WHILE ( M + 1 .LT. N )                                                           (  2-  3)
   20 ( 1)            .  ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
                     *.  ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                     *.  )
   21 ( 1)            .  I = ( M + N ) / 2
   22 ( 1)            .  IF ( X .LT. ARRAY ( I ) )                                                     (  4-  5)
   23 ( 2)            .  .  N = I
   24 ( 1)            .  ORIF ( X .GT. ARRAY ( I ) )                                                   (  6-  7)
   25 ( 2)            .  .  M = I
   26 ( 1)            .  ELSE
   27 ( 2)            .  .  LOOKUP = I
   28 ( 2)            .  .  M = I
   29 ( 2)            .  .  N = I + 1
   30 ( 1)            .  ENDIF
   31                 ENDWHILE
   32                 ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
                     *ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                     *)
   33                 IF ( X .NE. ARRAY ( M ) )                                                        (  8-  9)
   34 ( 1)            .  ERROR = .TRUE.
   35                 ELSE
   36 ( 1)            .  LOOKUP = M
   37                 ENDIF
   38                 FINAL ( .NOT. ERROR .AND. X .EQ. ARRAY ( LOOKUP ) .OR. ERROR )
   39                 RETURN
   40                 END
------------------------------------------------------------------------------------------------------------------
```

```
STATEMENT LISTING                SUBROUTINE FIND ( A, NN, F )                                   PAGE      1

  NO. LEVEL     LABEL   STATEMENT TEXT...                                                       DDPATHS
---------------------------------------------------------------------------------------------------------
    1                   SUBROUTINE FIND ( A, NN, F )                                            (  1)
    2           C       KOLN FIND
    3           C       HOARES FIND PROGRAM
    4                   LOGICAL PSORT
    5                   INTEGER A ( 1 ), NN, F
    6                   INTEGER I, J, M, N
    7                   INTEGER R, W, P, Q
    8                   INITIAL ( 1 .LE. F .AND. F .LE. NN )
    9                   M = 1
   10                   N = NN
   11                   WHILE ( M .LT. N )                                                      (  2-  3)
   12 ( 1)      C       .  M, N INVARIANTS
   13 ( 1)              .  ASSERT ( M .LE. F .AND. ( .ALL. P .IN. ( 1, M - 1 ) ( PSORT ( Q, M
                      *.   , NN, A, P, A, Q ) ) ) .AND. F .LE. N .AND. ( .ALL. P .IN. ( 1, N
                      *.   ) ( PSORT ( Q, N + 1, NN, A, P, A, Q ) ) ) )
   14 ( 1)              .  R = A ( F )
   15 ( 1)              .  I = M
   16 ( 1)              .  J = N
   17 ( 1)              .  WHILE ( I .LE. J )                                                   (  4-  5)
   18 ( 2)      C       .  .  I, J INVARIANTS
   19 ( 2)              .  .  ASSERT ( M .LE. I .AND. ( .ALL. P .IN. ( 1, I - 1 ) ( A ( P ) .LE.
                      *.         R ) ) .AND. J .LE. N .AND. ( .ALL. Q .IN. ( J + 1, NN ) ( R .LE.
                      *.      A ( Q ) ) ) )
   20 ( 2)              .  .  WHILE ( A ( I ) .LT. R )                                          (  6-  7)
   21 ( 3)      C       .  .  .  I INVARIANT
   22 ( 3)              .  .  .  ASSERT ( M .LE. I .AND. ( .ALL. P .IN. ( 1, I - 1 ) ( A ( P ) .LE.
                      *.            R ) ) )
   23 ( 3)              .  .  .  I = I + 1
   24 ( 2)              .  .  ENDWHILE
   25 ( 2)      C       .  .  I INVARIANT
   26 ( 2)              .  .  ASSERT ( M .LE. I .AND. ( .ALL. P .IN. ( 1, I - 1 ) ( A ( P ) .LE.
                      *.         R ) ) )
   27 ( 2)              .  .  WHILE ( R .LT. A ( J ) )                                          (  8-  9)
   28 ( 3)      C       .  .  .  J INVARIANT
   29 ( 3)              .  .  .  ASSERT ( J .LE. N .AND. ( .ALL. Q .IN. ( J + 1, NN ) ( R .LE. A (
                      *.            Q ) ) ) )
   30 ( 3)              .  .  .  J = J - 1
   31 ( 2)              .  .  ENDWHILE
   32 ( 2)              .  .  IF ( I .LE. J )                                                   ( 10- 11)
   33 ( 3)              .  .  .  W = A ( I )
   34 ( 3)              .  .  .  A ( I ) = A ( J )
   35 ( 3)              .  .  .  A ( J ) = W
   36 ( 3)              .  .  .  I = I + 1
   37 ( 3)              .  .  .  J = J - 1
   38 ( 2)              .  .  ENDIF
   39 ( 1)              .  ENDWHILE
   40 ( 1)      C       .  I, J INVARIANTS
   41 ( 1)              .  ASSERT ( M .LE. I .AND. ( .ALL. P .IN. ( 1, I - 1 ) ( A ( P ) .LE.
                      *.    R ) ) .AND. J .LE. N .AND. ( .ALL. Q .IN. ( J + 1, NN ) ( R .LE.
                      *.   A ( Q ) ) ) )
---------------------------------------------------------------------------------------------------------
```

```
STATEMENT LISTING                   SUBROUTINE FIND ( A, NN, F )                                    PAGE      2

  NO. LEVEL    LABEL   STATEMENT TEXT...                                                           DDPATHS
------------------------------------------------------------------------------------------------------------
   42 ( 1)             .  IF ( F .LE. J )                                                        ( 12- 13)
   43 ( 2)             .  .  N = J
   44 ( 1)             .  ELSE
   45 ( 2)             .  .  IF ( I .LE. F )                                                     ( 14- 15)
   46 ( 3)             .  .  .  M = I
   47 ( 2)             .  .  ELSE
   48 ( 3)             .  .  .  M = N
   49 ( 2)             .  .  ENDIF
   50 ( 1)             .  ENDIF
   51                  ENDWHILE
   52                  FINAL ( ( .ALL. P .IN. ( 1, F ) ( A ( P ) .LE. A ( F ) ) ) .AND. (
                     * .ALL. G .IN. ( F, NN ) ( A ( F ) .LE. A ( G ) ) ) )
   53                  RETURN
   54                  END
------------------------------------------------------------------------------------------------------------
```

```
STATEMENT LISTING                  REAL FUNCTION SQX ( X )                                      PAGE     1

  NO. LEVEL     LABEL   STATEMENT TEXT...                                                       DDPATHS..
----------------------------------------------------------------------------------------------------------
     1                  REAL FUNCTION SQX ( X )                                                 (  1)
     2            C     MODNAM(SQX)
     3            C
     4            C     SINGLE RECURSIVE VARIABLE
     5            C
     6                  REAL X, Y
     7                  INITIAL ( X .GE. 0.0 )
     8                  SQX = 0.0
     9                  IF ( X .GT. 0.0 )                                                       (  2-  3
    10 ( 1)             .  Y = 0.5 * X + 1.0
    11 ( 1)             .  WHILE ( Y - X / Y .GT. 0.000001 * Y )                                (  4-  5)
    12 ( 2)             .  .  ASSERT ( ( X .GE. 0.0 .AND. Y .GE. 0.0 ) .AND. Y * Y .GT. X )
    13 ( 2)             .  .  Y = 0.5 * ( Y + X / Y )
    14 ( 1)             .  ENDWHILE
    15 ( 1)             .  SQX = Y
    16                  ENDIF
    17                  FINAL ( SQX * SQX - X .LE. 0.000003 * X )
    18                  RETURN
    19                  END
----------------------------------------------------------------------------------------------------------
```

VERIFICATION OF TIMES

```
CONTENTS OF FILE/COMMAND PRIOR TO STARTUP OPERATION

NEW LIBRARY = TEMP.
START,LANGUAGE=IFTRAN.
BASIC.
FOR ALL MODULES.
STRUCTURAL.
PRINT,MODULE.
VCG,PATH=2,1,2
VCG,REPLACE.
B .GE. 1 = B .GT. 0
*END.
VCG,PATH=2,1,3
VCG,PATH=2,2,2
VCG,REPLACE.
Y .GT. 1 = Y .GE. 2
*END.
VCG,PATH=2,2,3
END FOR.
END.
```

```
VCG,PATH=2,1,2                      SUBROUTINE TIMES ( A, B, X )

 LINE                   PATH SOURCE TEXT

    1                   SUBROUTINE TIMES ( A, B, X )
   11                   INITIAL ( B .GE. 0 )
   12                   SUM = 0
   13                   Y = B
   15                   WHILE ( Y .NE. 0 )
   18 ( 1)              .  ASSERT ( SUM .EQ. A * ( B - Y ) .AND. Y .GE. 0 )
```

```
VERIFICATION CONDITION              SUBROUTINE TIMES ( A, B, X )

  LINE    VERIFICATION CONDITION

   11     B .GE. 0

           AND

   15     B .NE. 0

----------- IMPLIES ---------------------------------------------------

   18     0 + A .EQ. A * ( B - ( B - 1 ) ) .AND. B - 1 .GE. 0
```

```
VCG.PATH=2.1.2                         SUBROUTINE TIMES ( A, B, X )

CLAUSE     VERIFICATION CONDITION

     1     B .GT. 0

----------- IMPLIES ----------------------------------------------------

     2     B .GE. 1

RULE       B .GE. 1 = B .GT. 0

REPLACE    B .GE. 1 = B .GT. 0
```

```
VCG.REPLACE.                           SUBROUTINE TIMES ( A, B, X )

     PROOF OF VERIFICATION CONDITION COMPLETED
```

```
VCG.PATH=2.1.3                         SUBROUTINE TIMES ( A, B, X )
 LINE                  PATH SOURCE TEXT

   1                   SUBROUTINE TIMES ( A, B, X )
  11                   INITIAL ( B .GE. 0 )
  12                   SUM = 0
  13                   Y = B
  15                   WHILE ( Y .NE. 0 )
  21                   FINAL ( X .EQ. A * B )
```

```
VERIFICATION CONDITION                 SUBROUTINE TIMES ( A, B, X )
  LINE     VERIFICATION CONDITION

    11     B .GE. 0

             AND

    15     B .EQ. 0

----------- IMPLIES ----------------------------------------------------

    21     0 .EQ. A * B
```

```
VCG.PATH=2.1.3                         SUBROUTINE TIMES ( A, B, X )

     PROOF OF VERIFICATION CONDITION COMPLETED
```

```
VCG.PATH=2,2,2                            SUBROUTINE TIMES ( A, B, X )

 LINE                   PATH SOURCE TEXT

   15                   WHILE ( Y .NE. 0 )
   16 ( 1)              .   SUM = SUM + A
   17 ( 1)              .   Y = Y - 1
   18 ( 1)              .   ASSERT ( SUM .EQ. A * ( B - Y ) .AND. Y .GE. 0 )
   19                   ENDWHILE
   15                   WHILE ( Y .NE. 0 )
   18 ( 1)              .   ASSERT ( SUM .EQ. A * ( B - Y ) .AND. Y .GE. 0 )
```

```
VERIFICATION CONDITION                    SUBROUTINE TIMES ( A, B, X )

  LINE     VERIFICATION CONDITION

   15      Y .NE. 0

            AND

   18      SUM + A .EQ. A * ( B - ( Y - 1 ) ) .AND. Y - 1 .GE. 0

            AND

   15      Y - 1 .NE. 0

----------- IMPLIES ---------------------------------------------------

   18      SUM + A + A .EQ. A * ( B - ( Y - 1 - 1 ) ) .AND. Y - 1 - 1 .GE. 0
```

```
VCG.PATH=2,2,2                            SUBROUTINE TIMES ( A, B, X )

CLAUSE     VERIFICATION CONDITION

     1     Y .GT. 1

            AND

     2     - ( A * B ) + ( A * Y ) + SUM .EQ. 0

----------- IMPLIES ---------------------------------------------------

     3     Y .GE. 2

RULE       Y .GT. 1 = Y .GE. 2

REPLACE    Y .GT. 1 = Y .GE. 2
```

```
VCG.REPLACE.                              SUBROUTINE TIMES ( A, B, X )

     PROOF OF VERIFICATION CONDITION COMPLETED
```

```
VCG.PATH=2.2.3                          SUBROUTINE TIMES ( A, B, X )

 LINE                    PATH SOURCE TEXT

   15                    WHILE ( Y .NE. 0 )
   16 ( 1)               .   SUM = SUM + A
   17 ( 1)               .   Y = Y - 1
   18 ( 1)               .   ASSERT ( SUM .EQ. A * ( B - Y ) .AND. Y .GE. 0 )
   19                    ENDWHILE
   15                    WHILE ( Y .NE. 0 )
   21                    FINAL ( X .EQ. A * B )
```

```
VERIFICATION CONDITION             SUBROUTINE TIMES ( A, B, X )

  LINE    VERIFICATION CONDITION

    15    Y .NE. 0

           AND

    18    SUM + A .EQ. A * ( B - ( Y - 1 ) ) .AND. Y - 1 .GE. 0

           AND

    15    Y - 1 .EQ. 0

----------- IMPLIES ----------------------------------------------------

    21    SUM + A .EQ. A * B
```

```
VCG.PATH=2.2.3                          SUBROUTINE TIMES ( A, B, X )

    PROOF OF VERIFICATION CONDITION COMPLETED
```

VERIFICATION OF DIV

```
CONTENTS OF FILE/COMMAND PRIOR TO STARTUP OPERATION


NEW LIBRARY = TEMP.
START.LANGUAGE=IFTRAN.
BASIC.
FOR ALL MODULES.
STRUCTURAL.
PRINT.MODULE.
VCG.PATH=2.1.2
VCG.PATH=2.1.3
VCG. PATH=2.2.2
VCG.PATH=2.2.3
END FOR.
END.
```

```
VCG.PATH=2.1.2                        SUBROUTINE DIV ( A, B, Q, R )

 LINE                  PATH SOURCE TEXT

   1                   SUBROUTINE DIV ( A, B, Q, R )
  10                   INITIAL ( A .GE. 0 .AND. B .GE. 0 )
  11                   Q = 0
  12                   R = A
  13                   WHILE ( R .GE. B )
  14 ( 1)              .  ASSERT ( A .EQ. Q * B + R .AND. R .GE. 0 )
```

```
VERIFICATION CONDITION               SUBROUTINE DIV ( A, B, Q, R )

  LINE     VERIFICATION CONDITION

   10      A .GE. 0 .AND. B .GE. 0

            AND

   13      A .GE. B

----------- IMPLIES ---------------------------------------------------

   14      A .EQ. ( 0 * B ) + A .AND. A .GE. 0
```

```
VCG.PATH=2.1.2                        SUBROUTINE DIV ( A, B, Q, R )

      PROOF OF VERIFICATION CONDITION COMPLETED
```

```
VCG.PATH=2.1.3                          SUBROUTINE DIV ( A, B, Q, R )

 LINE                  PATH SOURCE TEXT

    1                  SUBROUTINE DIV ( A, B, Q, R )
   10                  INITIAL ( A .GE. 0 .AND. B .GE. 0 )
   11                  Q = 0
   12                  R = A
   13                  WHILE ( R .GE. B )
   18                  FINAL ( A .EQ. Q * B + R .AND. 0 .LE. R .AND. R .LT. B )
```

```
VERIFICATION CONDITION                  SUBROUTINE DIV ( A, B, Q, R )

  LINE    VERIFICATION CONDITION

   10     A .GE. 0 .AND. B .GE. 0

           AND

   13     A .LT. B

----------- IMPLIES --------------------------------------------------

   18     A .EQ. ( 0 * B ) + A .AND. 0 .LE. A .AND. A .LT. B
```

```
VCG.PATH=2.1.3                          SUBROUTINE DIV ( A, B, Q, R )

     PROOF OF VERIFICATION CONDITION COMPLETED
```

```
VCG. PATH=2,2,2                         SUBROUTINE DIV ( A, B, Q, R )

 LINE                  PATH SOURCE TEXT

   13                  WHILE ( R .GE. B )
   14 ( 1)             .  ASSERT ( A .EQ. Q * B + R .AND. R .GE. 0 )
   15 ( 1)             .  Q = Q + 1
   16 ( 1)             .  R = R - B
   17                  ENDWHILE
   13                  WHILE ( R .GE. B )
   14 ( 1)             .  ASSERT ( A .EQ. Q * B + R .AND. R .GE. 0 )
```

```
VERIFICATION CONDITION            SUBROUTINE DIV ( A, B, Q, R )

 LINE     VERIFICATION CONDITION

   13     R .GE. B

           AND

   14     A .EQ. ( Q * B ) + R .AND. R .GE. 0

           AND

   13     R - B .GE. B

----------- IMPLIES ----------------------------------------------------

   14     A .EQ. ( ( Q + 1 ) * B ) + R - B .AND. R - B .GE. 0
```

```
VCG, PATH=2,2,2                    SUBROUTINE DIV ( A, B, Q, R )


     PROOF OF VERIFICATION CONDITION COMPLETED
```

```
VCG, PATH=2,2,3                    SUBROUTINE DIV ( A, B, Q, R )

 LINE                  PATH SOURCE TEXT

   13                  WHILE ( R .GE. B )
   14 ( 1)             .  ASSERT ( A .EQ. Q * B + R .AND. R .GE. 0 )
   15 ( 1)             .  Q = Q + 1
   16 ( 1)             .  R = R - B
   17                  ENDWHILE
   13                  WHILE ( R .GE. B )
   18                  FINAL ( A .EQ. Q * B + R .AND. 0 .LE. R .AND. R .LT. B )
```

```
VERIFICATION CONDITION             SUBROUTINE DIV ( A, B, Q, R )

 LINE     VERIFICATION CONDITION

   13     R .GE. B

           AND

   14     A .EQ. ( Q * B ) + R .AND. R .GE. 0

           AND

   13     R - B .LT. B

----------- IMPLIES ----------------------------------------------------

   18     A .EQ. ( ( Q + 1 ) * B ) + R - B .AND. 0 .LE. R - B .AND. R - B .LT. B
```

```
VCG,PATH=2,2,3                    SUBROUTINE DIV ( A, B, Q, R )

     PROOF OF VERIFICATION CONDITION COMPLETED
```

VERIFICATION OF BINSCH

CONTENTS OF FILE/COMMAND PRIOR TO STARTUP OPERATION

```
START,LANGUAGE=IFTRAN.
BASIC.
MODULE=(FINSCH).
STRUCTURAL.
PRINT,MODULE.
FILENAME,LOG=OUTPUT.
VCG,PATH=2,1,2
VCG,PATH=2,1,3
VCG,PATH=2,3,8
VCG,PATH=2,3,9
VCG,REPLACE.
ERROR = .TRUE.
*END.
VCG,PATH=2,3,9
VCG,REPLACE.
ERROR = .FALSE.
*END.
VCG,PATH=3,2,4,2
VCG,AXIOM.
PX1 .LT. 0 = PX1 .LT. -1 .OR. PX1 .EQ. -1
*END.
VCG,AXIOM.
PX1 .IMP. PX2 .OR. PX3 = (PX1 .IMP. PX2) .OR. (PX1 .IMP. PX3)
*END.
VCG,PATH=4,2,5,6,2
VCG,REPLACE.
PX1 .LT. 0 = PX1 .LE. -1
*END.
VCG,REPLACE.
PX1 .AND. PX2 .LT. -1 .IMP. PX2 .LE. -1 .AND. PX3 =
PX1 .AND. PX2 .LT. -1 .IMP. PX3
*END.
VCG,REPLACE.
PX1 + PX2 .LE. 0 = - PX1 -PX2 .GE. 0
*END.
VCG,REPLACE.
PX1 .GE. 0 = PX1 .GT. 0 .OR. PX1 .EQ. 0
*END.
VCG,AXIOM,2.
VCG,PATH=4,2,5,7,2
VCG,PATH=3,2,4,3
VCG,AXIOM.
-((N+M)/2) + M .GE. -1 = -N+M .GE. -2
*END.
VCG,AXIOM.
PX1 .LT. -1 = PX1 .LE. -2
*END.
VCG,REPLACE.
-N + M .LE. -2 .AND. PX1 .AND. -N + M .GE. -2 =
-N + M .EQ. -2 .AND. PX1
*END.
VCG,REPLACE.
N = M+2
*END.
VCG,REPLACE.
((2 + M + M)/2) = M+1
*END.
VCG,PATH=4,2,5,6,3
VCG,REPLACE.
-N + ((N+M)/2) .GE. -1 =
-N + M .GE. -2
```

```
*END.
VCG,AXIOM,4.
VCG,REPLACE.
N = M+2
*END.
VCG,REPLACE.
((2 + M + M)/2) = M+1
*END.
VCG,REPLACE.
-X + ARRAY (( 1+M)) .LE. 0 =
X-ARRAY((1+M)) .GT. 0 .OR. X-ARRAY ((1+M)) .EQ. 0
*END.
VCG,AXIOM,2.
VCG,PATH=4,2,5,7,3
VCG,REPLACE.
X = ARRAY ((M+N)/2)
*END.
VCG,REPLACE.
SORTED(ARRAY,LENGTH)= (ARRAY(1) - ARRAY(I+1) .LT. 0 ) .AND. (I .GE. 1)
.AND. (I .LT. LENGTH)
*END.
VCG,REPLACE.
I = (M+N)/2
*END.
END.
```

```
VCG,PATH=2,1,2                    SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

 LINE           PATH SOURCE TEXT

   1            SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )
  15            INITIAL ( 1 .LT. LENGTH .AND. SORTED ( ARRAY, LENGTH ) .AND. (
               *ARRAY ( 1 ) .LE. X .AND. X .LT. ARRAY ( LENGTH ) ) )
  16            M = 1
  17            N = LENGTH
  18            ERROR = .FALSE.
  19            WHILE ( M + 1 .LT. N )
  20 ( 1)       .  ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
               *.  ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. FRROR )
               *.  )
```

```
VERIFICATION CONDITION          SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

  LINE    VERIFICATION CONDITION

   15     1 .LT. LENGTH .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( 1 ) .LF. X
           .AND. X .LT. ARRAY ( LENGTH )

           AND

   19     1 + 1 .LT. LENGTH
----------- IMPLIES -------------------------------------------------

   20     1 .LT. LENGTH .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( 1 ) .LF. X
           .AND. X .LT. ARRAY ( LENGTH ) .AND. .NOT. .FALSE.
```

```
VCG.PATH=2.1.2                    SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

     PROOF OF VERIFICATION CONDITION COMPLETED
```

```
VCG.PATH=2.1.3                    SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

 LINE             PATH SOURCE TEXT

   1              SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )
  15              INITIAL ( 1 .LT. LENGTH .AND. SORTED ( ARRAY, LENGTH ) .AND. (
                 *ARRAY ( 1 ) .LE. X .AND. X .LT. ARRAY ( LENGTH ) ) )
  16              M = 1
  17              N = LENGTH
  18              ERROR = .FALSE.
  19              WHILE ( M + 1 .LT. N )
  32              ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
                 *ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                 *)
```

```
VERIFICATION CONDITION            SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

  LINE    VERIFICATION CONDITION

   15     1 .LT. LENGTH .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( 1 ) .LE. X
           .AND. X .LT. ARRAY ( LENGTH )

           AND

   19     1 + 1 .GE. LENGTH

----------- IMPLIES --------------------------------------------------

   32     1 .LT. LENGTH .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( 1 ) .LE. X
           .AND. X .LT. ARRAY ( LENGTH ) .AND. .NOT. .FALSE.
```

```
VCG.PATH=2.1.3                    SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

     PROOF OF VERIFICATION CONDITION COMPLETED
```

```
VCG.PATH=2.3.8                    SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

 LINE             PATH SOURCE TEXT

  19              WHILE ( M + 1 .LT. N )
  32              ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
                 *ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                 *)
  33              IF ( X .NE. ARRAY ( M ) )
  38 ( 1)         .  FINAL ( .NOT. ERROR .AND. X .EQ. ARRAY ( LOOKUP ) .OR. ERROR )
```

```
VERIFICATION CONDITION        SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

 LINE    VERIFICATION CONDITION

  19     M + 1 .GE. N

          AND

  32     M .LT. N .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( M ) .LE. X .AND
         . X .LT. ARRAY ( N ) .AND. .NOT. ERROR

          AND

  33     X .NE. ARRAY ( M )

----------- IMPLIES -----------------------------------------------------

  38     ( .NOT. .TRUE. .AND. X .EQ. ARRAY ( LOOKUP ) ) .OR. .TRUE.
```

```
VCG.PATH=2.3.8                SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )


     PROOF OF VERIFICATION CONDITION COMPLETED
```

```
VCG.PATH=2.3.9                SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

 LINE            PATH SOURCE TEXT

  19             WHILE ( M + 1 .LT. N )
  32             ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
                *ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                *)
  33             IF ( X .NE. ARRAY ( M ) )
  38             FINAL ( .NOT. ERROR .AND. X .EQ. ARRAY ( LOOKUP ) .OR. ERROR )
```

```
VERIFICATION CONDITION        SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

 LINE    VERIFICATION CONDITION

  19     M + 1 .GE. N

          AND

  32     M .LT. N .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( M ) .LE. X .AND
         . X .LT. ARRAY ( N ) .AND. .NOT. ERROR

          AND

  33     X .EQ. ARRAY ( M )

----------- IMPLIES -----------------------------------------------------

  38     ( .NOT. ERROR .AND. X .EQ. ARRAY ( M ) ) .OR. ERROR
```

```
VCG.PATH=2.3.9             SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    X - ARRAY ( N ) .LT. 0

           AND

     2    .NOT. ERROR

           AND

     3    SORTED ( ARRAY , LENGTH )

           AND

     4    X - ARRAY ( M ) .EQ. 0

           AND

     5    - N + M .GE. -1

           AND

     6    - N + M .LT. 0

           AND

     7    - X + ARRAY ( M ) .LE. 0
---------- IMPLIES ---------------------------------------------

     8    ERROR .OR. ( .NOT. ERROR .AND. X - ARRAY ( M ) .EQ. 0 )

     ENTERED EXPRESSION

          ERROR = .TRUE.

RULE      ERROR = .TRUE.

REPLACE   ERROR = .TRUE.

REPLACE   ERROR = .TRUE.

REPLACE   ERROR = .TRUE.
```

```
VCG.REPLACE.               SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

     PROOF OF VERIFICATION CONDITION COMPLETED
```

```
VCG,PATH=2,3,9                  SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

LINE              PATH SOURCE TEXT

  19              WHILE ( M + 1 .LT. N )
  32              ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
                 *ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                 *)
  33              IF ( X .NE. ARRAY ( M ) )
  38              FINAL ( .NOT. ERROR .AND. X .EQ. ARRAY ( LOOKUP ) .OR. ERROR )
```

```
VERIFICATION CONDITION          SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

 LINE    VERIFICATION CONDITION

  19     M + 1 .GE. N

          AND

  32     M .LT. N .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( M ) .LE. X .AND
         . X .LT. ARRAY ( N ) .AND. .NOT. ERROR

          AND

  33     X .EQ. ARRAY ( M )
----------- IMPLIES --------------------------------------------------------

  38     ( .NOT. ERROR .AND. X .EQ. ARRAY ( M ) ) .OR. ERROR
```

```
VCG.PATH=2.3.9            SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    X - ARRAY ( N ) .LT. 0

           AND

     2    .NOT. ERROR

           AND

     3    SORTED ( ARRAY , LENGTH )

           AND

     4    X - ARRAY ( M ) .EQ. 0

           AND

     5    - N + M .GE. -1

           AND

     6    - N + M .LT. 0

           AND

     7    - X + ARRAY ( M ) .LE. 0
----------- IMPLIES -------------------------------------------------

     8    ERROR .OR. ( .NOT. ERROR .AND. X - ARRAY ( M ) .EQ. 0 )

     ENTERED EXPRESSION

          ERROR = .FALSE.

RULE      ERROR = .FALSE.

REPLACE   ERROR = .FALSE.

REPLACE   ERROR = .FALSE.

REPLACE   ERROR = .FALSE.
```

```
VCG.REPLACE.              SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

     PROOF OF VERIFICATION CONDITION COMPLETED
```

```
VCG.PATH=3.2.4.2              SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

LINE                PATH SOURCE TEXT

  19                WHILE ( M + 1 .LT. N )
  20 ( 1)            .  ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
                    *.  ARRAY ( M ) .LF. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                    *.  )
  21 ( 1)            .  I = ( M + N ) / 2
  22 ( 1)            .  IF ( X .LT. ARRAY ( I ) )
  23 ( 2)            .  .  N = I
  30 ( 1)            .  ENDIF
  31                ENDWHILE
  19                WHILE ( M + 1 .LT. N )
  20 ( 1)            .  ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
                    *.  ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                    *.  )
```

```
VERIFICATION CONDITION        SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

  LINE    VERIFICATION CONDITION

    19    M + 1 .LT. N

           AND

    20    M .LT. N .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( M ) .LE. X .AND
          . X .LT. ARRAY ( N ) .AND. .NOT. ERROR

           AND

    22    X .LT. ARRAY ( ( M + N ) / 2 )

           AND

    19    M + 1 .LT. ( M + N ) / 2
----------- IMPLIES ---------------------------------------------------

    20    M .LT. ( M + N ) / 2 .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( M )
           .LE. X .AND. X .LT. ARRAY ( ( M + N ) / 2 ) .AND. .NOT. ERROR
```

```
VCG,PATH=3,2,4,2          SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE     VERIFICATION CONDITION

     1     X - ARRAY ( N ) .LT. 0

            AND

     2     .NOT. ERROR

            AND

     3     SORTED ( ARRAY , LENGTH )

            AND

     4     X - ARRAY ( ( N + M ) / 2 ) .LT. 0

            AND

     5     - N + M .LT. -1

            AND

     6     - X + ARRAY ( M ) .LE. 0

            AND

     7     - ( ( N + M ) / 2 ) + M .LT. -1
----------- IMPLIES ---------------------------------------------------

     8     - ( ( N + M ) / 2 ) + M .LT. 0

    ENTERED EXPRESSION

           PX1 .LT. 0 = PX1 .LT. - 1 .OR. PX1 .EQ. - 1

RULE       PX1 .LT. 0 = ( PX1 .LT. -1 .OR. PX1 .EQ. -1 )

REPLACE    X - ARRAY ( N ) .LT. 0 = ( X - ARRAY ( N ) .LT. -1 .OR. X - ARRAY ( N
           ) .EQ. -1 )

REPLACE    X - ARRAY ( ( N + M ) / 2 ) .LT. 0 = ( X - ARRAY ( ( N + M ) / 2 ) .LT
           . -1 .OR. X - ARRAY ( ( N + M ) / 2 ) .EQ. -1 )

REPLACE    - ( ( N + M ) / 2 ) + M .LT. 0 = - ( ( N + M ) / 2 ) + M .LT. -1 .OR.
           - ( ( N + M ) / 2 ) + M .EQ. -1

     SAVED AS AXIOM   1
```

```
VCG.AXIOM.                    SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    ( X - ARRAY ( N ) .LT. -1 .OR. X - ARRAY ( N ) .EQ. -1 )

           AND

     2    .NOT. ERROR

           AND

     3    SORTED ( ARRAY , LENGTH )

           AND

     4    ( X - ARRAY ( ( N + M ) / 2 ) .LT. -1 .OR. X - ARRAY ( ( N + M ) / 2 )
           .EQ. -1 )

           AND

     5    - N + M .LT. -1

           AND

     6    - X + ARRAY ( M ) .LE. 0

           AND

     7    - ( ( N + M ) / 2 ) + M .LT. -1
----------- IMPLIES ---------------------------------------------------

     8    - ( ( N + M ) / 2 ) + M .LT. -1 .OR. - ( ( N + M ) / 2 ) + M .EQ. -1

     ENTERED EXPRESSION

          PX1 .IMP. PX2 .OR. PX3 = ( PX1 .IMP. PX2 ) .OR. ( PX1 .IMP. PX3 )

RULE      ( PX1 .IMP. PX2 .OR. PX3 ) = ( ( PX1 .IMP. PX2 ) .OR. ( PX1 .IMP. PX3
          ) )

REPLACE   ( ( X - ARRAY ( N ) .LT. -1 .OR. X - ARRAY ( N ) .EQ. -1 ) .AND. .NOT.
           ERROR .AND. SORTED ( ARRAY , LENGTH ) .AND. ( X - ARRAY ( ( N + M ) /

           2 ) .LT. -1 .OR. X - ARRAY ( ( N + M ) / 2 ) .EQ. -1 ) .AND. - N + M

          .LT. -1 .AND. - X + ARRAY ( M ) .LE. 0 .AND. - ( ( N + M ) / 2 ) + M .

          LT. -1 .IMP. - ( ( N + M ) / 2 ) + M .LT. -1 .OR. - ( ( N + M ) / 2 )

          + M .EQ. -1 ) = ( ( ( X - ARRAY ( N ) .LT. -1 .OR. X - ARRAY ( N ) .EQ

          . -1 ) .AND. .NOT. ERROR .AND. SORTED ( ARRAY , LENGTH ) .AND. ( X - A

          RRAY ( ( N + M ) / 2 ) .LT. -1 .OR. X - ARRAY ( ( N + M ) / 2 ) .EQ. -

          1 ) .AND. - N + M .LT. -1 .AND. - X + ARRAY ( M ) .LE. 0 .AND. - ( ( N

           + M ) / 2 ) + M .LT. -1 .IMP. - ( ( N + M ) / 2 ) + M .LT. -1 ) .OR.

          ( ( X - ARRAY ( N ) .LT. -1 .OR. X
SAVED AS AXIOM   2
```

```
VCG.AXIOM.                      SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

     PROOF OF VERIFICATION CONDITION COMPLETED
     USING THE FOLLOWING AXIOMS        1    2
```

```
VCG.PATH=4.2.5.6.2              SUBROUTINE BINSCH ( ARRAY, LENGTH. X. LOOKUP, ERROR )

 LINE             PATH SOURCE TEXT

   19             WHILE ( M + 1 .LT. N )
   20 ( 1)        .  ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
                 *.  ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                 *.  )
   21 ( 1)        .  I = ( M + N ) / 2
   22 ( 1)        .  IF ( X .LT. ARRAY ( I ) )
   24             ORIF ' X .GT. ARRAY ( I ) )
   25 ( 1)        .  M = I
   30             ENDIF
   31             ENDWHILE
   19             WHILE ( M + 1 .LT. N )
   20             ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY. LENGTH ) ) .AND. ( (
                 *ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                 *)
```

```
VERIFICATION CONDITION          SUBROUTINE BINSCH ( ARRAY, LENGTH. X. LOOKUP, ERROR )

  LINE    VERIFICATION CONDITION

   19     M + 1 .LT. N

           AND

   20     M .LT. N .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( M ) .LE. X .AND
          . X .LT. ARRAY ( N ) .AND. .NOT. ERROR

           AND

   22     X .GE. ARRAY ( ( M + N ) / 2 )

           AND

   24     X .GT. ARRAY ( ( M + N ) / 2 )

           AND

   19     ( ( M + N ) / 2 ) + 1 .LT. N
----------- IMPLIES ---------------------------------------------------
   20     ( M + N ) / 2 .LT. N .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( ( M
           + N ) / 2 ) .LE. X .AND. X .LT. ARRAY ( N ) .AND. .NOT. ERROR
```

```
VCG.PATH=4.2.5.6.2          SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    X - ARRAY ( N ) .LT. 0

           AND

     2    .NOT. ERROR

           AND

     3    SORTED ( ARRAY , LENGTH )

           AND

     4    X - ARRAY ( ( N + M ) / 2 ) .GT. 0

           AND

     5    - N + M .LT. -1

           AND

     6    - X + ARRAY ( M ) .LE. 0

           AND

     7    - N + ( ( N + M ) / 2 ) .LT. -1
----------- IMPLIES ---------------------------------------------------

     8    - N + ( ( N + M ) / 2 ) .LT. 0

           AND

     9    - X + ARRAY ( ( N + M ) / 2 ) .LE. 0

     ENTERED EXPRESSION

          PX1 .LT. 0 = PX1 .LE. - 1

RULE      PX1 .LT. 0 = PX1 .LE. -1

REPLACE   X - ARRAY ( N ) .LT. 0 = X - ARRAY ( N ) .LE. -1

REPLACE   - N + ( ( N + M ) / 2 ) .LT. 0 = - N + ( ( N + M ) / 2 ) .LE. -1
```

```
VCG.REPLACE.                        SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    X - ARRAY ( N ) .LE. -1

           AND

     2    .NOT. ERROR

           AND

     3    SORTED ( ARRAY , LENGTH )

           AND

     4    X - ARRAY ( ( N + M ) / 2 ) .GT. 0

           AND

     5    - N + M .LT. -1

           AND

     6    - X + ARRAY ( M ) .LE. 0

           AND

     7    - N + ( ( N + M ) / 2 ) .LT. -1
----------- IMPLIES ------------------------------------------------
     8    - N + ( ( N + M ) / 2 ) .LE. -1

           AND

     9    - X + ARRAY ( ( N + M ) / 2 ) .LE. 0

     ENTERED EXPRESSION

          PX1 .AND. PX2 .LT. - 1 .IMP. PX2 .LE. - 1 .AND. PX3 = PX1 .AND. PX2 .LT. - 1 .IM
          P. PX3

RULE      ( PX1 .AND. PX2 .LT. -1 .IMP. PX2 .LE. -1 .AND. PX3 ) = ( PX1 .AND. PX
          2 .LT. -1 .IMP. PX3 )

REPLACE   ( X - ARRAY ( N ) .LE. -1 .AND. .NOT. ERROR .AND. SORTED ( ARRAY , LEN
          GTH ) .AND. X - ARRAY ( ( N + M ) / 2 ) .GT. 0 .AND. - N + M .LT. -1 .

          AND. - X + ARRAY ( M ) .LE. 0 .AND. - N + ( ( N + M ) / 2 ) .LT. -1 .I

          MP. - N + ( ( N + M ) / 2 ) .LE. -1 .AND. - X + ARRAY ( ( N + M ) / 2

          ) .LE. 0 ) = ( X - ARRAY ( N ) .LE. -1 .AND. .NOT. ERROR .AND. SORTED

          ( ARRAY , LENGTH ) .AND. X - ARRAY ( ( N + M ) / 2 ) .GT. 0 .AND. - N

          + M .LT. -1 .AND. - X + ARRAY ( M ) .LE. 0 .AND. - N + ( ( N + M ) / 2

           ) .LT. -1 .IMP. - X + ARRAY ( ( N + M ) / 2 ) .LE. 0 )
```

```
VCG.REPLACE.                SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    X - ARRAY ( N ) .LE. -1

           AND

     2    .NOT. ERROR

           AND

     3    SORTED ( ARRAY , LENGTH )

           AND

     4    X - ARRAY ( ( N + M ) / 2 ) .GT. 0

           AND

     5    - N + M .LT. -1

           AND

     6    - X + ARRAY ( M ) .LE. 0

           AND

     7    - N + ( ( N + M ) / 2 ) .LT. -1
----------- IMPLIES ---------------------------------------------------

     8    - X + ARRAY ( ( N + M ) / 2 ) .LE. 0

     ENTERED EXPRESSION

          PX1 + PX2 .LE. 0 = - PX1 - PX2 .GE. 0

RULE      PX1 + PX2 .LE. 0 = - PX1 - PX2 .GE. 0

REPLACE   - X + ARRAY ( M ) .LE. 0 = - - X - ARRAY ( M ) .GE. 0

REPLACE   - X + ARRAY ( ( N + M ) / 2 ) .LE. 0 = - - X - ARRAY ( ( N + M ) / 2 )
           .GE. 0
```

```
VCG,REPLACE.                SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE     VERIFICATION CONDITION

     1     X - ARRAY ( N ) .LE. -1

            AND

     2     .NOT. ERROR

            AND

     3     SORTED ( ARRAY , LENGTH )

            AND

     4     X - ARRAY ( ( N + M ) / 2 ) .GT. 0

            AND

     5     - N + M .LT. -1

            AND

     6     X - ARRAY ( M ) .GE. 0

            AND

     7     - N + ( ( N + M ) / 2 ) .LT. -1
----------- IMPLIES -------------------------------------------------

     8     X - ARRAY ( ( N + M ) / 2 ) .GE. 0

     ENTERED EXPRESSION

           PX1 .GE. 0 = PX1 .GT. 0 .OR. PX1 .EQ. 0

RULE       PX1 .GE. 0 = ( PX1 .GT. 0 .OR. PX1 .EQ. 0 )

REPLACE    X - ARRAY ( M ) .GE. 0 = ( X - ARRAY ( M ) .GT. 0 .OR. X - ARRAY ( M )
            .EQ. 0 )

REPLACE    X - ARRAY ( ( N + M ) / 2 ) .GE. 0 = X - ARRAY ( ( N + M ) / 2 ) .GT.
           0 .OR. X - ARRAY ( ( N + M ) / 2 ) .EQ. 0
```

```
VCG.REPLACE.                SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    X - ARRAY ( N ) .LE. -1

           AND

     2    .NOT. ERROR

           AND

     3    SORTED ( ARRAY , LENGTH )

           AND

     4    X - ARRAY ( ( N + M ) / 2 ) .GT. 0

           AND

     5    - N + M .LT. -1

           AND

     6    ( X - ARRAY ( M ) .GT. 0 .OR. X - ARRAY ( M ) .EQ. 0 )

           AND

     7    - N + ( ( N + M ) / 2 ) .LT. -1
----------- IMPLIES ----------------------------------------------------

     8    X - ARRAY ( ( N + M ) / 2 ) .GT. 0 .OR. X - ARRAY ( ( N + M ) / 2 ) .E
          Q. 0

     USING AXIOM  2

RULE      ( PX1 .IMP. PX2 .OR. PX3 ) = ( ( PX1 .IMP. PX2 ) .OR. ( PX1 .IMP. PX3
          ) )

REPLACE   ( X - ARRAY ( N ) .LE. -1 .AND. .NOT. ERROR .AND. SORTED ( ARRAY , LEN
          GTH ) .AND. X - ARRAY ( ( N + M ) / 2 ) .GT. 0 .AND. - N + M .LT. -1 .

          AND. ( X - ARRAY ( M ) .GT. 0 .OR. X - ARRAY ( M ) .EQ. 0 ) .AND. - N

          + ( ( N + M ) / 2 ) .LT. -1 .IMP. X - ARRAY ( ( N + M ) / 2 ) .GT. 0 .

          OR. X - ARRAY ( ( N + M ) / 2 ) .EQ. 0 ) = ( ( X - ARRAY ( N ) .LE. -1

           .AND. .NOT. ERROR .AND. SORTED ( ARRAY , LENGTH ) .AND. X - ARRAY ( (

           N + M ) / 2 ) .GT. 0 .AND. - N + M .LT. -1 .AND. ( X - ARRAY ( M ) .G

          T. 0 .OR. X - ARRAY ( M ) .EQ. 0 ) .AND. - N + ( ( N + M ) / 2 ) .LT.

          -1 .IMP. X - ARRAY ( ( N + M ) / 2 ) .GT. 0 ) .OR. ( X - ARRAY ( N ) .

          LE. -1 .AND. .NOT. ERROR .AND. SORTED ( ARRAY , LENGTH ) .AND. X - ARR

          AY ( ( N + M ) / 2 ) .GT. 0 .AND. - N + M .LT. -1 .AND. ( X - ARRAY (

          M
```

```
VCG.AXIOM.2.            SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

     PROOF OF VERIFICATION CONDITION COMPLETED
     USING THE FOLLOWING AXIOMS        2
```

```
VCG.PATH=4.2.5.7.2            SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

 LINE              PATH SOURCE TEXT

   19              WHILE ( M + 1 .LT. N )
   20 ( 1)         .  ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
                  *.  ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                  *.  )
   21 ( 1)         .  I = ( M + N ) / 2
   22 ( 1)         .  IF ( X .LT. ARRAY ( I ) ;
   24 ( 1)         .  ORIF ( X .GT. ARRAY ( I ) )
   26              ELSE
   27 ( 1)         .  LOOKUP = 1
   28 ( 1)         .  M = 1
   29 ( 1)         .  N = I + 1
   30              ENDIF
   31              ENDWHILE
   19              WHILE ( M + 1 .LT. N )
   20              ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
                  *ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                  *)
```

```
VERIFICATION CONDITION       SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

  LINE    VERIFICATION CONDITION

   19     M + 1 .LT. N

           AND

   20     M .LT. N .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( M ) .LE. X .AND
          . X .LT. ARRAY ( N ) .AND. .NOT. ERROR

           AND

   22     X .GE. ARRAY ( ( M + N ) / 2 )

           AND

   24     X .LE. ARRAY ( ( M + N ) / 2 )

           AND

   19     ( ( M + N ) / 2 ) + 1 .LT. ( ( M + N ) / 2 ) + 1
----------- IMPLIES --------------------------------------------------

   20     ( M + N ) / 2 .LT. ( ( M + N ) / 2 ) + 1 .AND. SORTED ( ARRAY , LENGTH
           ) .AND. ARRAY ( ( M + N ) / 2 ) .LE. X .AND. X .LT. ARRAY ( ( ( M + N
           ) / 2 ) + 1 ) .AND. .NOT. ERROR
```

```
VCG.PATH=4.2.5.7.2          SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

     PROOF OF VERIFICATION CONDITION COMPLETED
```

```
VCG.PATH=3.2.4.3              SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

 LINE           PATH SOURCE TEXT

   19            WHILE ( M + 1 .LT. N )
   20 ( 1)       .  ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
                *.  ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                *.  )
   21 ( 1)       .  I = ( M + N ) / 2
   22 ( 1)       .  IF ( X .LT. ARRAY ( I ) )
   23 ( 2)       .  .  N = I
   30 ( 1)       .  ENDIF
   31            ENDWHILE
   19            WHILE ( M + 1 .LT. N )
   32            ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
                *ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
                *)
```

```
VERIFICATION CONDITION       SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

  LINE     VERIFICATION CONDITION

    19     M + 1 .LT. N

            AND

    20     M .LT. N .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( M ) .LE. X .AND
           . X .LT. ARRAY ( N ) .AND. .NOT. ERROR

            AND

    22     X .LT. ARRAY ( ( M + N ) / 2 )

            AND

    19     M + 1 .GE. ( M + N ) / 2
----------- IMPLIES --------------------------------------------------

    32     M .LT. ( M + N ) / 2 .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( M )
            .LE. X .AND. X .LT. ARRAY ( ( M + N ) / 2 ) .AND. .NOT. ERROR
```

```
VCG.PATH=3.2.4.3          SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    X - ARRAY ( N ) .LT. 0

           AND

     2    .NOT. ERROR

           AND

     3    SORTED ( ARRAY , LENGTH )

           AND

     4    X - ARRAY ( ( N + M ) / 2 ) .LT. 0

           AND

     5    - N + M .LT. -1

           AND

     6    - X + ARRAY ( M ) .LE. 0

           AND

     7    - ( ( N + M ) / 2 ) + M .GE. -1
---------- IMPLIES ------------------------------------------------
     8    - ( ( N + M ) / 2 ) + M .LT. 0

     ENTERED EXPRESSION

          - ( ( N + M ) / 2 ) + M .GE. - 1 = - N + M .GE. - 2

RULE      - ( ( N + M ) / 2 ) + M .GE. -1 = - N + M .GE. -2

REPLACE   - ( ( N + M ) / 2 ) + M .GE. -1 = - N + M .GE. -2

     SAVED AS AXIOM   3
```

```
VCG.AXIOM.                   SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE     VERIFICATION CONDITION

     1     X - ARRAY ( N ) .LT. 0

            AND

     2     .NOT. ERROR

            AND

     3     SORTED ( ARRAY , LENGTH )

            AND

     4     X - ARRAY ( ( N + M ) / 2 ) .LT. 0

            AND

     5     - N + M .LT. -1

            AND

     6     - X + ARRAY ( M ) .LE. 0

            AND

     7     - N + M .GE. -2
----------- IMPLIES ---------------------------------------------------

     8     - ( ( N + M ) / 2 ) + M .LT. 0

     ENTERED EXPRESSION

           PX1 .LT. - 1 = PX1 .LE. - 2

RULE       PX1 .LT. -1 = PX1 .LE. -2

REPLACE    - N + M .LT. -1 = - N + M .LE. -2

     SAVED AS AXIOM   4
```

```
VCG.AXIOM.                        SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    X - ARRAY ( N ) .LT. 0

           AND

     2    .NOT. ERROR

           AND

     3    SORTED ( ARRAY , LENGTH )

           AND

     4    X - ARRAY ( ( N + M ) / 2 ) .LT. 0

           AND

     5    - N + M .LE. -2

           AND

     6    - X + ARRAY ( M ) .LE. 0

           AND

     7    - N + M .GE. -2
----------- IMPLIES -------------------------------------------------

     8    - ( ( N + M ) / 2 ) + M .LT. 0

     ENTERED EXPRESSION

          - N + M .LE. - 2 .AND. PX1 .AND. - N + M .GE. - 2 = - N + M .EQ. - 2 .AND. PX1

RULE      ( - N + M .EQ. -2 .AND. PX1 ) = ( - N + M .EQ. -2 .AND. PX1 )

     NO REPLACEMENTS PERFORMED

     ENTERED EXPRESSION

          N = M + 2

RULE      N = ( 2 + M )

REPLACE   N = 2 + M

REPLACE   N = 2 + M

REPLACE   N = ( 2 + M )

REPLACE   N = ( 2 + M )

REPLACE   N = 2 + M
```

```
VCG,REPLACE.             SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    .NOT. ERROR

           AND

     2    SORTED ( ARRAY , LENGTH )

           AND

     3    X - ARRAY ( 2 + M ) .LT. 0

           AND

     4    X - ARRAY ( ( 2 + M + M ) / 2 ) .LT. 0

           AND

     5    - X + ARRAY ( M ) .LE. 0
----------- IMPLIES --------------------------------------------------

     6    - ( ( 2 + M + M ) / 2 ) + M .LT. 0

     ENTERED EXPRESSION

          ( ( 2 + M + M ) / 2 ) = M + 1

RULE      ( ( 2 + M + M ) / 2 ) = ( 1 + M )

REPLACE   ( 2 + M + M ) / 2 = 1 + M

REPLACE   ( ( 2 + M + M ) / 2 ) = ( 1 + M )
```

```
VCG,REPLACE.              SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

     PROOF OF VERIFICATION CONDITION COMPLETED
     USING THE FOLLOWING AXIOMS         3    4
```

```
VCG,PATH=4,2,5,6,3         SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )
 LINE          PATH SOURCE TEXT

   19          WHILE ( M + 1 .LT. N )
   20 ( 1)     .  ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
              *.  ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
              *.  )
   21 ( 1)     .  I = ( M + N ) / 2
   22 ( 1)     .  IF ( X .LT. ARRAY ( I ) )
   24          ORIF ( X .GT. ARRAY ( I ) )
   25 ( 1)     .  M = I
   30          ENDIF
   31          ENDWHILE
   19          WHILE ( M + 1 .LT. N )
   32          ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
              *ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
              *)
```

```
VERIFICATION CONDITION        SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

 LINE    VERIFICATION CONDITION

   19    M + 1 .LT. N

          AND

   20    M .LT. N .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( M ) .LE. X .AND
         . X .LT. ARRAY ( N ) .AND. .NOT. ERROR

          AND

   22    X .GE. ARRAY ( ( M + N ) / 2 )

          AND

   24    X .GT. ARRAY ( ( M + N ) / 2 )

          AND

   19    ( ( M + N ) / 2 ) + 1 .GE. N
----------- IMPLIES ---------------------------------------------------

   32    ( M + N ) / 2 .LT. N .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( ( M
          + N ) / 2 ) .LE. X .AND. X .LT. ARRAY ( N ) .AND. .NOT. ERROR
```

```
VCG,PATH=4,2,5,6,3        SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    X - ARRAY ( N ) .LT. 0

           AND

     2    .NOT. ERROR

           AND

     3    SORTED ( ARRAY , LENGTH )

           AND

     4    X - ARRAY ( ( N + M ) / 2 ) .GT. 0

           AND

     5    - N + M .LT. -1

           AND

     6    - X + ARRAY ( M ) .LE. 0

           AND

     7    - N + ( ( N + M ) / 2 ) .GE. -1
----------- IMPLIES ------------------------------------------------------
     8    - N + ( ( N + M ) / 2 ) .LT. 0

           AND

     9    - X + ARRAY ( ( N + M ) / 2 ) .LE. 0

    ENTERED EXPRESSION

          - N + ( ( N + M ) / 2 ) .GE. - 1 = - N + M .GE. - 2

RULE      - N + ( ( N + M ) / 2 ) .GE. -1 = - N + M .GE. -2

REPLACE   - N + ( ( N + M ) / 2 ) .GE. -1 = - N + M .GE. -2
```

```
VCG.REPLACE.            SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE     VERIFICATION CONDITION

     1     X - ARRAY ( N ) .LT. 0

            AND

     2     .NOT. ERROR

            AND

     3     SORTED ( ARRAY , LENGTH )

            AND

     4     X - ARRAY ( ( N + M ) / 2 ) .GT. 0

            AND

     5     - N + M .LT. -1

            AND

     6     - X + ARRAY ( M ) .LE. 0

            AND

     7     - N + M .GE. -2
----------- IMPLIES -----------------------------------------------
     8     - N + ( ( N + M ) / 2 ) .LT. 0

            AND

     9     - X + ARRAY ( ( N + M ) / 2 ) .LE. 0

     USING AXIOM  4

RULE       PX1 .LT. -1 = PX1 .LE. -2

REPLACE    - N + M .LT. -1 = - N + M .LE. -2
```

```
VCG.AXIOM.4.               SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    X - ARRAY ( N ) .LT. 0

           AND

     2    .NOT. ERROR

           AND

     3    SORTED ( ARRAY , LENGTH )

           AND

     4    X - ARRAY ( ( N + M ) / 2 ) .GT. 0

           AND

     5    - N + M .LE. -2

           AND

     6    - X + ARRAY ( M ) .LE. 0

           AND

     7    - N + M .GE. -2
----------- IMPLIES --------------------------------------------------
     8    - N + ( ( N + M ) / 2 ) .LT. 0

           AND

     9    - X + ARRAY ( ( N + M ) / 2 ) .LE. 0

     ENTERED EXPRESSION

          N = M + 2

RULE      N = ( 2 + M )

REPLACE   N = 2 + M

REPLACE   N = 2 + M

REPLACE   N = ( 2 + M )

REPLACE   N = ( 2 + M )

REPLACE   N = ( 2 + M )

REPLACE   N = 2 + M

REPLACE   N = 2 + M
```

```
VCG.REPLACE.              SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

    1     .NOT. ERROR

           AND

    2     SORTED ( ARRAY , LENGTH )

           AND

    3     X - ARRAY ( 2 + M ) .LT. 0

           AND

    4     X - ARRAY ( ( 2 + M + M ) / 2 ) .GT. 0

           AND

    5     - X + ARRAY ( M ) .LE. 0
----------- IMPLIES --------------------------------------------------

    6     - M + ( ( 2 + M + M ) / 2 ) .LT. 2

           AND

    7     - X + ARRAY ( ( 2 + M + M ) / 2 ) .LE. 0

    ENTERED EXPRESSION

          ( ( 2 + M + M ) / 2 ) = M + 1

RULE      ( ( 2 + M + M ) / 2 ) = ( 1 + M )

REPLACE   ( 2 + M + M ) / 2 = 1 + M

REPLACE   ( ( 2 + M + M ) / 2 ) = 1 + M

REPLACE   ( 2 + M + M ) / 2 = 1 + M
```

```
VCG.REPLACE.                          SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE     VERIFICATION CONDITION

     1     .NOT. ERROR

            AND

     2     SORTED ( ARRAY , LENGTH )

            AND

     3     X - ARRAY ( 2 + M ) .LT. 0

            AND

     4     X - ARRAY ( 1 + M ) .GT. 0

            AND

     5     - X + ARRAY ( M ) .LE. 0
----------- IMPLIES -----------------------------------------------

     6     - X + ARRAY ( 1 + M ) .LE. 0

     ENTERED EXPRESSION

           - X + ARRAY ( ( 1 + M ) ) .LE. 0 = X - ARRAY ( ( 1 + M ) ) .GT. 0 .OR. X - ARRAY
            ( ( 1 + M ) ) .EQ. 0

RULE       - X + ARRAY ( 1 + M ) .LE. 0 = ( X - ARRAY ( 1 + M ) .GT. 0 .OR. X - A
           RRAY ( 1 + M ) .EQ. 0 )

REPLACE    - X + ARRAY ( 1 + M ) .LE. 0 = X - ARRAY ( 1 + M ) .GT. 0 .OR. X - ARR
           AY ( 1 + M ) .EQ. 0
```

```
VCG.REPLACE.                SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE     VERIFICATION CONDITION

     1     .NOT. ERROR

            AND

     2     SORTED ( ARRAY , LENGTH )

            AND

     3     X - ARRAY ( 2 + M ) .LT. 0

            AND

     4     X - ARRAY ( 1 + M ) .GT. 0

            AND

     5     - X + ARRAY ( M ) .LE. 0
----------- IMPLIES -------------------------------------------------

     6     X - ARRAY ( 1 + M ) .GT. 0 .OR. X - ARRAY ( 1 + M ) .EQ. 0

     USING AXIOM  2

RULE       ( PX1 .IMP. PX2 .OR. PX3 ) = ( ( PX1 .IMP. PX2 ) .OR. ( PX1 .IMP. PX3
           ) )

REPLACE    ( .NOT. ERROR .AND. SORTED ( ARRAY , LENGTH ) .AND. X - ARRAY ( 2 + M
           ) .LT. 0 .AND. X - ARRAY ( 1 + M ) .GT. 0 .AND. - X + ARRAY ( M ) .LE.

            0 .IMP. X - ARRAY ( 1 + M ) .GT. 0 .OR. X - ARRAY ( 1 + M ) .EQ. 0 )

           = ( ( .NOT. ERROR .AND. SORTED ( ARRAY , LENGTH ) .AND. X - ARRAY ( 2

           + M ) .LT. 0 .AND. X - ARRAY ( 1 + M ) .GT. 0 .AND. - X + ARRAY ( M )

           .LE. 0 .IMP. X - ARRAY ( 1 + M ) .GT. 0 ) .OR. ( .NOT. ERROR .AND. SOR

           TED ( ARRAY , LENGTH ) .AND. X - ARRAY ( 2 + M ) .LT. 0 .AND. X - ARRA

           Y ( 1 + M ) .GT. 0 .AND. - X + ARRAY ( M ) .LE. 0 .IMP. X - ARRAY ( 1

           + M ) .EQ. 0 ) )
```

```
VCG.AXIOM.2.                SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

     PROOF OF VERIFICATION CONDITION COMPLETED
     USING THE FOLLOWING AXIOMS        4    2
```

```
VCG.PATH=4.2.5.7.3           SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

LINE          PATH SOURCE TEXT

  19          WHILE ( M + 1 .LT. N )
  20 ( 1)     .  ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
             *.  ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
             *.  )
  21 ( 1)     .  I = ( M + N ) / 2
  22 ( 1)     .  IF ( X .LT. ARRAY ( I ) )
  24 ( 1)     .  ORIF ( X .GT. ARRAY ( I ) )
  26          ELSE
  27 ( 1)     .  LOOKUP = 1
  28 ( 1)     .  M = I
  29 ( 1)     .  N = I + 1
  30          ENDIF
  31          ENDWHILE
  19          WHILE ( M + 1 .LT. N )
  32          ASSERT ( ( M .LT. N .AND. SORTED ( ARRAY, LENGTH ) ) .AND. ( (
             *ARRAY ( M ) .LE. X .AND. X .LT. ARRAY ( N ) ) .AND. .NOT. ERROR )
             *)
```

```
VERIFICATION CONDITION       SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

  LINE    VERIFICATION CONDITION

    19    M + 1 .LT. N

           AND

    20    M .LT. N .AND. SORTED ( ARRAY , LENGTH ) .AND. ARRAY ( M ) .LE. X .AND
          . X .LT. ARRAY ( N ) .AND. .NOT. ERROR

           AND

    22    X .GE. ARRAY ( ( M + N ) / 2 )

           AND

    24    X .LE. ARRAY ( ( M + N ) / 2 )

           AND

    19    ( ( M + N ) / 2 ) + 1 .GE. ( ( M + N ) / 2 ) + 1
----------- IMPLIES ---------------------------------------------------

    32    ( M + N ) / 2 .LT. ( ( M + N ) / 2 ) + 1 .AND. SORTED ( ARRAY , LENGTH
           ) .AND. ARRAY ( ( M + N ) / 2 ) .LE. X .AND. X .LT. ARRAY ( ( ( M + N
           ) / 2 ) + 1 ) .AND. .NOT. ERROR
```

```
VCG.PATH=4.2.5.7.3        SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    X - ARRAY ( N ) .LT. 0

           AND

     2    .NOT. ERROR

           AND

     3    SORTED ( ARRAY , LENGTH )

           AND

     4    X - ARRAY ( ( N + M ) / 2 ) .EQ. 0

           AND

     5    - N + M .LT. -1

           AND

     6    - X + ARRAY ( M ) .LE. 0
---------- IMPLIES ---------------------------------------------------
     7    X - ARRAY ( 1 + ( ( N + M ) / 2 ) ) .LT. 0

           AND

     8    - X + ARRAY ( ( N + M ) / 2 ) .LE. 0

     ENTERED EXPRESSION

          X = ARRAY ( ( M + N ) / 2 )

RULE      X = ARRAY ( ( N + M ) / 2 )

REPLACE   X = ARRAY ( ( N + M ) / 2 )

REPLACE   X = ARRAY ( ( N + M ) / 2 )

REPLACE   X = ARRAY ( ( N + M ) / 2 )

REPLACE   X = ARRAY ( ( N + M ) / 2 )

REPLACE   X = ARRAY ( ( N + M ) / 2 )
```

```
VCG,REPLACE.                    SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    .NOT. ERROR

           AND

     2    SORTED ( ARRAY , LENGTH )

           AND

     3    - ARRAY ( N ) + ARRAY ( ( N + M ) / 2 ) .LT. 0

           AND

     4    - N + M .LT. -1

           AND

     5    - ARRAY ( ( N + M ) / 2 ) + ARRAY ( M ) .LE. 0
----------- IMPLIES ----------------------------------------------

     6    ARRAY ( ( N + M ) / 2 ) - ARRAY ( 1 + ( ( N + M ) / 2 ) ) .LT. 0

     ENTERED EXPRESSION

          SORTED ( ARRAY , LENGTH ) = ( ARRAY ( I ) - ARRAY ( I + 1 ) .LT. 0 ) .AND. ( I .
          GE. 1 ) .AND. ( I .LT. LENGTH )

RULE      SORTED ( ARRAY , LENGTH ) = ( I .GE. 1 .AND. ARRAY ( I ) - ARRAY ( 1 +
           I ) .LT. 0 .AND. I - LENGTH .LT. 0 )

REPLACE   SORTED ( ARRAY , LENGTH ) = I .GE. 1 .AND. ARRAY ( I ) - ARRAY ( 1 + I
           ) .LT. 0 .AND. I - LENGTH .LT. 0
```

```
VCG.REPLACE.            SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

CLAUSE    VERIFICATION CONDITION

     1    I .GE. 1

           AND

     2    ARRAY ( I ) - ARRAY ( 1 + I ) .LT. 0

           AND

     3    I - LENGTH .LT. 0

           AND

     4    .NOT. ERROR

           AND

     5    - ARRAY ( N ) + ARRAY ( ( N + M ) / 2 ) .LT. 0

           AND

     6    - N + M .LT. -1

           AND

     7    - ARRAY ( ( N + M ) / 2 ) + ARRAY ( M ) .LE. 0
----------- IMPLIES ---------------------------------------------------

     8    ARRAY ( ( N + M ) / 2 ) - ARRAY ( 1 + ( ( N + M ) / 2 ) ) .LT. 0

     ENTERED EXPRESSION

          I = ( M + N ) / 2

RULE      I = ( ( N + M ) / 2 )

REPLACE   I = ( N + M ) / 2

REPLACE   I = ( N + M ) / 2

REPLACE   I = ( ( N + M ) / 2 )

REPLACE   I = ( ( N + M ) / 2 )
```

```
VCG.REPLACE.            SUBROUTINE BINSCH ( ARRAY, LENGTH, X, LOOKUP, ERROR )

     PROOF OF VERIFICATION CONDITION COMPLETED
```

VERIFICATION CONDITIONS FOR SQX

```
VCG.PATH=2,1,3                      REAL FUNCTION SQX ( X )

 LINE                 PATH SOURCE TEXT

    1                 REAL FUNCTION SQX ( X )
    7                 INITIAL ( X .GE. 0.0 )
    8                 SQX = 0.0
    9                 IF ( X .GT. 0.0 )
   17                 FINAL ( SQX * SQX - X .LE. 0.000003 * X )
```

```
VERIFICATION CONDITION              REAL FUNCTION SQX ( X )

  LINE    VERIFICATION CONDITION

     7    X .GE. 0.0

           AND

     9    X .LE. 0.0

----------- IMPLIES --------------------------------------------------

    17    ( 0.0 * 0.0 ) - X .LE. 0.000003 * X
```

```
VCG.PATH=2,1,3                      REAL FUNCTION SQX ( X )

     PROOF OF VERIFICATION CONDITION COMPLETED
```

```
VCG.PATH=3,1,2,4            REAL FUNCTION SQX ( X )

 LINE            PATH SOURCE TEXT

    1            REAL FUNCTION SQX ( X )
    7            INITIAL ( X .GE. 0.0 )
    8            SQX = 0.0
    9            IF ( X .GT. 0.0 )
   10 ( 1)       .  Y = 0.5 * X + 1.0
   11 ( 1)       .  WHILE ( Y - X / Y .GT. 0.000001 * Y )
   12 ( 2)       .  .  ASSERT ( ( X .GE. 0.0 .AND. Y .GE. 0.0 ) .AND. Y * Y .GT. X )
```

```
VERIFICATION CONDITION              REAL FUNCTION SQX ( X )

 LINE     VERIFICATION CONDITION

    7     X .GE. 0.0

           AND

    9     X .GT. 0.0

           AND

   11     ( 0.5 * X ) + 1.0 - ( X / ( ( 0.5 * X ) + 1.0 ) ) .GT. 0.000001 * ( (
          0.5 * X ) + 1.0 )

----------- IMPLIES ----------------------------------------------------

   12     X .GE. 0.0 .AND. ( 0.5 * X ) + 1.0 .GE. 0.0 .AND. ( ( 0.5 * X ) + 1.0
          ) * ( ( 0.5 * X ) + 1.0 ) .GT. X
```

```
VCG.PATH=3.1.2.4                    REAL FUNCTION SQX ( X )

CLAUSE    VERIFICATION CONDITION

    1     - ( 0.0000005 * X ) + ( 0.5 * X ) - ( X / ( 1.0 + ( 0.5 * X ) ) ) .GT.
           1.000001

           AND

    2     X .GT. 0.0

----------- IMPLIES ----------------------------------------------------

    3     - X + ( 0.5 * X ) + ( 0.5 * X ) + ( 0.25 * X * X ) .GT. 1.0

           AND

    4     X .GE. 0.0

           AND

    5     0.5 * X .GE. 1.0
```

```
VCG.PATH=3.1.2.5                    REAL FUNCTION SQX ( X )

 LINE                 PATH SOURCE TEXT

   1                  REAL FUNCTION SQX ( X )
   7                  INITIAL ( X .GE. 0.0 )
   8                  SQX = 0.0
   9                  IF ( X .GT. 0.0 )
  10 ( 1)             .  Y = 0.5 * X + 1.0
  11 ( 1)             .  WHILE ( Y - X / Y .GT. 0.000001 * Y )
  17 ( 1)             .  FINAL ( SQX * SQX - X .LE. 0.000003 * X )
```

```
VERIFICATION CONDITION           REAL FUNCTION SQX ( X )

 LINE     VERIFICATION CONDITION

    7     X .GE. 0.0

           AND

    9     X .GT. 0.0

           AND

   11     ( 0.5 * X ) + 1.0 - ( X / ( ( 0.5 * X ) + 1.0 ) ) .LE. 0.000001 * ( (
          0.5 * X ) + 1.0 )
----------- IMPLIES -----------------------------------------------------

   17     ( ( ( 0.5 * X ) + 1.0 ) * ( ( 0.5 * X ) + 1.0 ) ) - X .LE. 0.000003 *
          X
```

```
VCG.PATH=3.1.2.5                 REAL FUNCTION SQX ( X )

CLAUSE    VERIFICATION CONDITION

    1     - ( 0.0000005 * X ) + ( 0.5 * X ) - ( X / ( 1.0 + ( 0.5 * X ) ) ) .LE.
           1.000001

           AND

    2     X .GT. 0.0
----------- IMPLIES -----------------------------------------------------

    3     - ( 0.000003 * X ) + ( 0.5 * X ) + ( 0.5 * X ) + ( 0.25 * X * X ) - X
          .LE. 1.0
```

```
VCG.PATH=2.4.4                REAL FUNCTION SQX ( X )

 LINE            PATH SOURCE TEXT

   11            WHILE ( Y - X / Y .GT. 0.000001 * Y )
   12 ( 1)       .  ASSERT ( ( X .GE. 0.0 .AND. Y .GE. 0.0 ) .AND. Y * Y .GT. X )
   13 ( 1)       .  Y = 0.5 * ( Y + X / Y )
   14            ENDWHILE
   11            WHILE ( Y - X / Y .GT. 0.000001 * Y )
   12 ( 1)       .  ASSERT ( ( X .GE. 0.0 .AND. Y .GE. 0.0 ) .AND. Y * Y .GT. X )
```

```
VERIFICATION CONDITION            REAL FUNCTION SQX ( X )

 LINE    VERIFICATION CONDITION

   11    Y - ( X / Y ) .GT. 0.000001 * Y

          AND

   12    X .GE. 0.0 .AND. Y .GE. 0.0 .AND. Y * Y .GT. X

          AND

   11    ( 0.5 * ( Y + ( X / Y ) ) ) - ( X / ( 0.5 * ( Y + ( X / Y ) ) ) ) .GT.
          0.000001 * 0.5 * ( Y + ( X / Y ) )
---------- IMPLIES ----------------------------------------------------

   12    X .GE. 0.0 .AND. 0.5 * ( Y + ( X / Y ) ) .GE. 0.0 .AND. 0.5 * ( Y + (
         X / Y ) ) * 0.5 * ( Y + ( X / Y ) ) .GT. X
```

```
VCG,PATH=2,4,4                     REAL FUNCTION SQX ( X )

CLAUSE    VERIFICATION CONDITION

     1    X .GE. 0.0

           AND

     2    - ( 0.0000005 * Y ) - ( 0.0000005 * ( X / Y ) ) + ( 0.5 * Y ) + ( 0.5
          * ( X / Y ) ) - ( X / ( ( 0.5 * Y ) + ( 0.5 * ( X / Y ) ) ) ) .GT. 0

           AND

     3    - ( 0.000001 * Y ) + Y - ( X / Y ) .GT. 0

           AND

     4    - X + ( Y * Y ) .GT. 0

           AND

     5    Y .GE. 0.0
---------- IMPLIES ----------------------------------------------------

     6    ( 0.5 * Y ) + ( 0.5 * ( X / Y ) ) .GE. 0.0

           AND

     7    - X + ( 0.25 * Y * Y ) + ( 0.25 * Y * ( X / Y ) ) + ( 0.25 * ( X / Y )
          * Y ) + ( 0.25 * ( X / Y ) * ( X / Y ) ) .GT. 0
```

```
VCG.PATH=2.4.5                REAL FUNCTION SQX ( X )

 LINE           PATH SOURCE TEXT

   11           WHILE ( Y - X / Y .GT. 0.000001 * Y )
   12 ( 1)      .  ASSERT ( ( X .GE. 0.0 .AND. Y .GE. 0.0 ) .AND. Y * Y .GT. X )
   13 ( 1)      .  Y = 0.5 * ( Y + X / Y )
   14           ENDWHILE
   11           WHILE ( Y - X / Y .GT. 0.000001 * Y )
   17           FINAL ( SQX * SQX - X .LE. 0.000003 * X )
```

```
VERIFICATION CONDITION            REAL FUNCTION SQX ( X )

  LINE     VERIFICATION CONDITION

    11     Y - ( X / Y ) .GT. 0.000001 * Y

            AND

    12     X .GE. 0.0 .AND. Y .GE. 0.0 .AND. Y * Y .GT. X

            AND

    11     ( 0.5 * ( Y + ( X / Y ) ) ) - ( X / ( 0.5 * ( Y + ( X / Y ) ) ) ) .LE.
            0.000001 * 0.5 * ( Y + ( X / Y ) )
----------- IMPLIES ---------------------------------------------------

    17     ( 0.5 * ( Y + ( X / Y ) ) * 0.5 * ( Y + ( X / Y ) ) ) - X .LE. 0.0n000
           3 * X
```

```
VCG.PATH=2.4.5                    REAL FUNCTION SQX ( X )

CLAUSE     VERIFICATION CONDITION

     1     X .GE. 0.0

            AND

     2     - ( 0.0000005 * Y ) - ( 0.0000005 * ( X / Y ) ) + ( 0.5 * Y ) + ( 0.5
           * ( X / Y ) ) - ( X / ( ( 0.5 * Y ) + ( 0.5 * ( X / Y ) ) ) ) .LE. 0

            AND

     3     - ( 0.000001 * Y ) + Y - ( X / Y ) .GT. 0

            AND

     4     - X + ( Y * Y ) .GT. 0

            AND

     5     Y .GE. 0.0
----------- IMPLIES ---------------------------------------------------

     6     - ( 0.000003 * X ) - X + ( 0.25 * Y * Y ) + ( 0.25 * Y * ( X / Y ) ) +
            ( 0.25 * ( X / Y ) * Y ) + ( 0.25 * ( X / Y ) * ( X / Y ) ) .LE. 0
```

REFERENCES

1. J. P. Benson, R. A. Melton and W. R. Wisehart, Research Plan for Reliable Software, General Research Corporation CR-1-676, November 1975.

2. J. P. Benson, R. A. Melton and W. R. Wisehart, BMDATC Reliable Software Study, General Research Corporation CR-2-676, December 1975.

3. R. W. Floyd, "Assigning Meanings to Programs," Proceedings Symposium in Applied Mathematics, American Mathematical Society, Vol. 19, 1967, pp. 19-32.

4. R. W. Floyd, "The Verifying Compiler," Computer Science Research Review, Carnegie-Mellon University, 1967, pp. 18-19.

5. J. King, A Program Verifier, Ph.D. Thesis, Carnegie-Mellon University, 1969.

6. L. P. Deutsch, An Interactive Program Verifier, Ph.D. Thesis, University of California, Berkeley, 1973.

7. B. Elspas, et al., A Verification System for JOVIAL-J3 Programs, Stanford Research Institute (for RADC) Report No. RADC-TR-76-58, March 1976.

8. R. L. London, "A View of Program Verification," Proceedings International Conference on Reliable Software, April 1975, pp. 534-545.

9. D. Luckham and N. Suzuki, Automatic Program Verification V, Stanford University Computer Science Department Report No. STAN-CS-76-549, March 1976.

10. L. Osterweil and L. D. Fosdick, "Some Experience with DAVE - A FORTRAN Program Analyzer," AFIPS National Computer Conference, Vol. 45, 1976, pp. 909-915.

11. Verification and Validation for Terminal Defense Program Software: The Development of a Software Error Theory to Classify and Detect Software Errors, Logicon HR-74012, May 1974.

REFERENCES (Contd.)

12. K. Jensen and N. Wirth, PASCAL User Manual and Report, Springer-Verlag, New York, 1975.

13. D. S. Alberts, "The Economics of Software Quality Assurance," Proceedings of the AFIPS National Computer Conference, Vol. 45, 1976, pp. 433-442.

14. K. E. Kilbride, Compiler to Write and Implement Computers (CWIC), System Development Corporation Report No. TM-HU-225/000/00, 1 September 1976.

15. O. Lecarme and G. V. Bochmann, "A (Truly) Useable and Portable Compiler Writing System," Information Processing 74, North-Holland Publishing Company, 1974.

16. E. R. Buley, et al., Advanced Software Quality Assurance, Mid-Study Technical Report, General Research Corporation CR-3-720, October 1976.

17. O. Lecarme and G. V. Bochmann, A Compiler Writing System User's Manual, Research Group for Systems and Programming Languages Document #57, Department d'Informatique, Universite de Montreal, December 1974.

18. M. E. Dyer, et al., Software Design Specification, SREP Methodology, TRW Systems Group Report No. 27332-6921-014, October 1975.

19. P. Ward, Un Systeme d'ecriture de compilateurs a analyse syntaxique descendante, Manual d'utilisation, Document #70, Department d'Informatique, Universite de Montreal, October 1975.

20. BMDATC Data Processing Standards, ARC Data Processing User's Guide, Vol. III, BMD Advanced Technology Center Report No. TM-HU-212/003/004, July 1976.

21. L. J. Osterweil and L. D. Fosdick, Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation, and Error Detection, University of Colorado Computer Science Department Report No. CU-CS-055-74, September 1974.

22. B. Wegbriet, "The Synthesis of Loop Predicates," Communications of the ACM, Vol. 17, No. 2, February 1974, pp. 102-112.

23. RXVP User's Manual, Program Validation Project, General Research Corporation, July 1976.

REFERENCES (Contd.)

24. C. A. R. Hoare, "Proof of a Program: FIND," Communications of the ACM, Vol. 14, No. 1, January 1971. pp. 39-44.

25. Per Brinch-Hansen, "Concurrent PASCAL Report," Information Science, California Institute of Technology, June 1975.

26. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," Communications of the ACM, Vol. 17, No. 10, October 1974, pp. 549-557.

27. E. W. Dijkstra, "Cooperating Sequential Processes," Programming Languages, Academic Press, New York, 1968, pp. 43-112.

28. V. G. Cerf, Multiprocessors, Semaphores and a Graph Model of Computation, Ph.D. Thesis, University of California, Los Angeles, Computer Science Department Report No. UCLA-10, April 1972, pp. 14-110.

29. H. Vantilborgh and A. van Lamsweerde, "On an Extension of Dijkstra's Semaphore Primitives," Information Processing Letters, 1, North Holland Publishing Co., New York, October 1972, pp. 181-186.

30. P. J. Courtois, F. Heymans and D. L. Parnas, "Concurrent Control with Readers and Writers," Communications of the ACM, Vol. 14, No. 10, October 1971, pp. 667-668.

31. E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," Communications of the ACM, Vol. 8, No. 9, September 1965, p. 569.

32. E. W. Dijkstra, "Information Streams Sharing a Finite Buffer," Information Processing Letters, 1, North Holland Publishing Co., New York, 1972, pp. 178-180.

DISTRIBUTION

Copies

Director
Ballistic Missile Defense
Advanced Technology Center
Attn: ATC-P 2
P.O. Box 1500
Huntsville, AL 35807

Ballistic Missile Defense Program Office
Attn: DACS-BMT, Attn: Dr. Merwin 1
Attn: DACS-BMS 1
Commonwealth Building
1300 Wilson Blvd.
Arlington, VA 22209

Commander
Ballistic Missile Defense System Command
Attn: BMDSC-C *
P.O. Box 1500
Huntsville, AL 35807

Defense Documentation Center
Cameron Station
Alexandria, VA 22314 2

System Development Corporation
4810 Bradford Blvd. N. W.
Huntsville, AL 35805
Attn: BMDATC Technical Library 10

Computer Sciences Corporation
Huntsville Operations
515 Sparkman Drive, N. W.
Huntsville, AL 35805
Attn: Mr. Pete Belford 1

Aeronutronic Ford
Communications System Division
3900 Welsch Road
Willow Grove, PA 19090 1

*Transmittal letter only

DISTRIBUTION (Contd.)

Copies

TRW Systems
7702 Governors Drive, West
Huntsville, AL 35805
Attn: Mr. Larry Marker 1

System Development Corporation
4810 Bradford Blvd. N.W.
Huntsville, AL 35805
Attn: Mr. Robert Covelli 1

University of Alabama in Huntsville,
Computer Science Program
P.O. Box 1247
Huntsville, AL 35807
Attn: Dr. Pei Hsia
Room RI/M340 1